```
[an error occurred while processing this directive]
```

Lisp as a Second Language

Chapter 1 Functional Style

Draft mars 11, 1997

1. Lists as the main data structure
2. Abstraction and application as dual mechanisms
3. Conditionals as control structure
4. Recursion as main control structure
5. Binding as a way to retain intermediate results
6. Functions as first class objects
7. Functions as arguments
8. Functions as results
9. Generality as aim
10. Combinators as function builders
11. More on generality as aim
12. Parameters as superfluous
13. Definitions made
14. Literature references made
15. Glossary references made
16. Text references made

This chapter introduces the functional style of programming with lists as the sole data structure and recursion as the main control structure. It will present the basics of Lisp in the form of examples that you will find reused throughout the book.

Lists as the main data structure

The primary data type of Lisp is the *list*. It is notated as an open parenthesis followed by zero, or more elements, and a closing parenthesis. To give an example, the following list could be used as a representation for a note that has a duration of 2 time-units, a pitch of C (MIDI [1] key number 60) and maximum loudness:

```
(note 2 60 1)
```

Consider the choice of this primitive musical object here as an arbitrary one, we just need one such object in our examples. We will see that the choice of primitives is a very important one, based on esthetic grounds -it expresses your idea of the atomic musical object and its parameters- and on pragmatic grounds

-it will determine which operations can be expressed easily and elegantly, and which are more difficult to express or even impossible.

There is one special list, the empty list, notated as `()` or `nil`. The elements of a list can be symbols, called atoms, (like `note` or `60`) or they can be lists. Thus a collection of notes can be represented as a list of lists:

```
((note 1 60 1)(note 2 61 0.7)(note 1 55 1))
```

If we wish to express control over the timing structure in the representation, ordering a collection of notes, we could form sequential and parallel structures. This way of specifying the time relations between objects is an alternative to the use of absolute start times found in most composition systems. It makes explicit the time-relation between its elements. `Sequential` indicates that its elements are in succeeding order, one after the other (like notes in a melody), `parallel` indicates that its elements are simultanuous (like in a chord). The first example shows a sequential object, the second a nesting of a parallel and sequential object. As such we can construct more elaborated structured or compound musical objects.

```
(sequential (note 1 60 1)(note 2 61 0.7))
```

```
(parallel (sequential (note 1 60 1)
(note 2 63 1))
(note 4 55 0.5))
```

A piano-roll notation of the second musical object is given in Figure 1. Figure 2 visualises its tree structure as a *part-of hierarchy*: `parallel`, at the top, has two parts (a `sequential` structure and a `note`) and `sequential`, in turn, has two notes as parts.

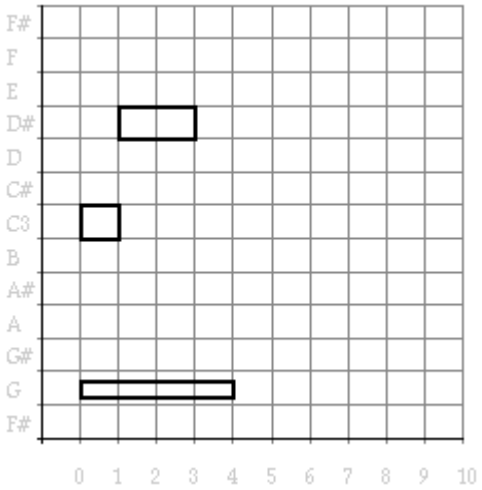

Figure 1. Piano-roll notation of the example.

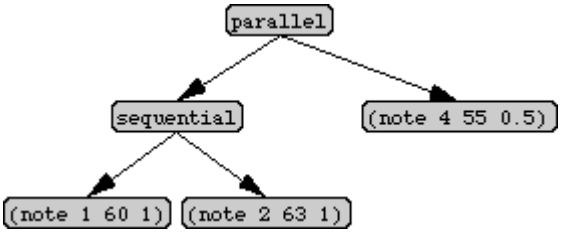

Figure 2. Example data structure.

Note that the words like `note`, `sequential` and `parallel` do not have any intrinsic meaning here, since they are not being build-in Lisp functions. They are just used as arbitrary symbols, signalling our intention with the data in that list. In Common Lisp there are of course other data types available like strings, arrays, hash tables, etc. which sometimes are more appropriate then lists. They will be discussed in later chapters, here we will restrict ourselves to lists.

Abstraction and application as dual mechanisms

The very heart of any functional programming language consists of a pair of dual constructs, the first of which is called *application*. It is the action of 'calling' a function on one or more arguments. The syntactical form of an application is the name of the function followed by its arguments, together enclosed by parentheses (prefix notation). In the example below the function `+` is applied to the arguments `1` and `3`. The arrow (->) points to the result of the evaluation of an expression.

```
(+ 1 3) -> 4
```

```
(first '(note 1 60 1)) -> note
```

The second expression applies the function `first` to one of our note objects. Note that the list is preceded by a quote (`'`). In this way we make it a constant data list. It prevents the Lisp interpreter from recognizing `(note 1 60 1)` as an application of a function `note` to the arguments 1, 60 and 1. The result of evaluating the expression is the atom `note`.

There are two selector functions that take lists apart: `first` and `rest`.[2] There is one constructor function for building lists, called `cons`. Below some examples using these functions.

```
(rest '(note 1 60 1)) -> (1 60 1)
```

```
(first (rest '(note 1 60 1))) -> 1
```

```
(cons 'note '(1 60 1)) -> (note 1 60 1)
```

Note that in these examples atoms, like `note`, have to be quoted (to prevent them from being interpreted as a variable, something we will discuss shortly). Numbers are always constant and do not need to be quoted.

There are more Lisp primitives available for the construction of lists (e.g., `append` that concatenates several lists and `list` which builds a list out of several elements at once). However, they can all be written by the programmer using only `cons`. Below three ways of constructing the same note data structure:

```
(append '(note 1) '(60 1)) -> (note 1 60 1)
```

```
(list 'note 1 60 1) -> (note 1 60 1)
```

```
(cons 'note (cons 1 (cons 60 (cons 1 nil)))) -> (note 1 60 1)
```

Application should be used as the only means to pass information to a function. This ensures that the behavior of functions is not dependant upon the context of its call. In other words, the result of applying a function is only dependent on the value of its the argument: it is garanteed to give the same result, independent of where in the program code this function was evoked. This is a very central and useful notion in functional programming.[3]

The second central notion is called functional *abstraction* and transforms an expression (a piece of program text) into a function. This can be done with the lisp construct `defun`. `Defun` is a *special form*, which means that it uses a non-standard indiosyncratic syntax consisting of the name of the function, a list of zero or more arguments, a documentation string describing the function (text enclosed by double quotes (`"`)), and the *body* of the function. The function below is a constant function named `example` without arguments that will just return, as a result, a simple constant musical structure. We will use this function a lot in the next paragraphs:

```
(defun example ()
"Return a constant musical object"
'(sequential
```

```
(parallel (note 1 60 1)

(note 2 63 1))

(note 4 55 0.5)))


(example) ->

(sequential (parallel (note 1 60 1)

(note 2 63 1))

(note 4 55 0.5))


(draw (example)) =>
```

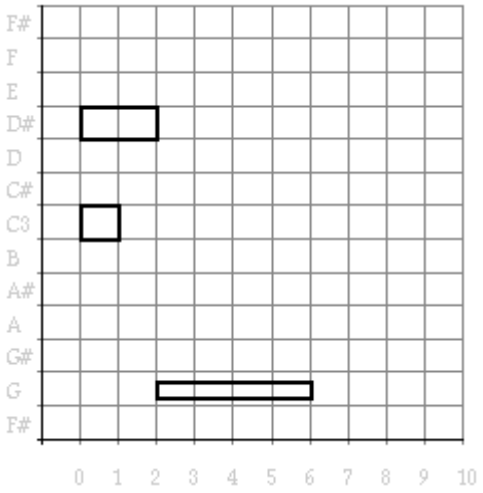

Figure 3. The pianoroll representation of the example.

We will use the function `draw` to generate graphical piano-roll representation[4] of our examples, a function you will be able to write yourself somewhat later in this chapter.

The documentation string that we add to all our functions is always be accessible, and can be obtained from all lisp functions. The function `documentation` takes the name of a function and its type (in this chapter always a function), and returns the documentation string describing it.

```
(documentation 'list 'function) ->

"Return a list containing arguments as its elements."
```

The documentation string of our newly defined function `example` is accessible as well:

```
(documentation 'example 'function) ->

"Return a constant musical object"
```

Documentation strings can form the basis of simple tools for documentation or for sophisticated auto documenting programs, as we will see later. For now, we will move on and define a number of functions that have one argument and select a specific element out of a list.[5]

In the first example below a function called `second- element` is defined which has one parameter called `list`. In the definition the parameter `list` is said to be abstracted from the body. Only when `second-element` is applied to an actual argument does the body becomes 'concrete' in the sense that it 'knows' what the parameter `list` stands for, so that its value can be calculated. Its body is the application of the functions `first` and `rest` to this parameter.

```
(defun second-element (list)

"Return second element of list"

(first (rest list)))


(defun third-element (list)

"Return third element of list"

(first (rest (rest list))))


(defun fourth-element (list)

"Return fourth element of list"

(first (rest (rest (rest list)))))


(second-element '(c d e f g a b)) -> d


(third-element '(c d e f g a b)) -> e


(fourth-element '(c d e f g a b)) -> f
```

A data structure always is accompagnied by functions that can access them, so-called *selector* functions, and a function that can make a new one, named a *constructor* function. Together they are referred to as a

data abstraction layer, since the use (acessing and making them) and the actual form of the data (a list in our case) are clearly separated. When changing the data type (e.g., to an array) we only have to change this set of functions without affecting the workings of our other programs. We will now describe a constructor function and some selector functions for our musical object `note` making use of our list access functions defined above.

```
(defun make-note (duration pitch loudness)
"Return a note data structure"
(list 'note duration pitch loudness))

(defun duration (note)
"Return the duration of note"
(second-element note))

(defun pitch (note)
"Return the pitch of note"
(third-element note))

(defun loudness (note)
"Return the amplitude of note"
(fourth-element note))

(duration '(note 3 60 1)) -> 3

(pitch '(note 3 60 1)) -> 60

(make-note 3 60 1) -> (note 3 60 1)
```

In the same way we could program a set of selector and constructor functions as a data abstraction layer for sequential and parallel structures.

```
(defun make-sequential (musical-object-1 musical-object-2)
"Return a sequential data structure"
(list 'sequential musical-object-1 musical-object-2))
```

```
(defun make-parallel (musical-object-1 musical-object-2)
"Return a parallel data structure"
(list 'parallel musical-object-1 musical-object-2))

(defun first-structure-of (musical-object)
"Return the first element of a musical object"
(second-element musical-object))

(defun second-structure-of (complex-structure)
"Return the second element of a musical object"
(third-element complex-structure))

(defun structural-type-of (musical-object)
"Return a type of a musical object"
(first musical-object))

(structural-type-of (example)) -> sequential

(second-structure-of (example)) -> (note 4 55 0.5)
```

This is a good moment to play around with these expressions in your Lisp interpreter, and get aquainted with the mechanism of evaluating Lisp expressions and defining functions (if you not already did this!).

It is not always desired to pass values for all arguments of a function, for instance, when default values would be good enough. For this Common Lisp provides control in the form of *lambda-list keywords*. Optional parameters to a function (that will be assigned a default value when missing in the function call) can be defined by means of the `&optional` lambda-list keyword, followed by one or more lists each containing an argument name and its default value.[6]Below we redefine our constructor `make- note` using optional parameters.

```
(defun make-note (&optional (duration 1) (pitch 60) (loudness 1))
"Return a note data structure"
(list 'note duration pitch loudness))

(make-note 2 61) -> (note 2 61 1)
```

```
(make-note) -> (note 1 60 1)
```

Another useful lambda-list keyword is `&rest`. It is followed by one parameter that collects all the arguments of the function is a list;

```
(defun make-note (&rest args)
"Return a note data structure"
(cons 'note args))

(make-note 2 61 1) -> (note 2 61 1)
```

A third lambda-list keyword is `&key`, [7] with which the order in which the arguments have to be passed to a function is not fixed anymore. The syntax is the same as the optional keyword: one or more lists containing the argument name and its default value. When calling such a function, every argument has to be preceded by a keyword indicating for which parameter the value following it is intended. Below another rewrite of the function `make- note` using keywords, one that we will stick to for a while.

```
(defun make-note (&key (duration 1) (pitch 60) (loudness 1))
"Return a note data structure"
(list 'note duration pitch loudness))

(make-note :duration 2 :pitch 61) -> (note 2 61 1)

(make-note :pitch 62 :duration 2) -> (note 2 62 1)
```

Combinations of different lambda-list keywords can generate some confusion. For the time being, a good rule is to use only one at a time in a function definition.

We now can use these new definitions to construct some useful note transformations:

```
;;; utility functions

(defun clip (min value max)
"Return the value, clipped within [min,max]"
```

```
(cond ((< value min) min)
((> value max) max)
(t value)))

(defun transpose-pitch (pitch interval)
"Return pitch increased by interval"
(+ pitch interval))

(defun mirror-pitch (pitch center)
"Return pitch mirrored around center"
(- center (- pitch center)))

;;; general note transforms

(defun transpose-note (note interval)
"Return note transposed by interval"
(make-note :duration (duration note)
:pitch (transpose-pitch (pitch note) interval)
:loudness (loudness note)))

(defun limit-loudness (note low high)
"Return note with clipped loudness"
(make-note :duration (duration note)
:pitch (pitch note)
:loudness (clip low (loudness note) high)))

(defun mirror-note (note center)
"Return note mirrored around center"
(make-note :duration (duration note)
:pitch (mirror-pitch (pitch note) center)
:loudness (loudness note)))

;;; dedicated note transforms
```

```
(defun transpose-note-semitone (note)

"Return note transposed by semi-tone"

(transpose-note note 1))


(defun mirror-note-around-middle-c (note)

"Return note mirrored around middle-c"

(mirror-note note 60))


(limit-loudness '(note 1 57 .2) 0.5 1) -> (note 1 57 0.5)


(transpose-note-semitone '(note 1 60 1)) -> (note 1 61 1)


(mirror-note-around-middle-c '(note 1 57 1)) -> (note 1 63 1)
```

Since this is a larger code fragment, we use comments to visually group portions of the code. Comments are text preceeded by a `;` (a semicolon). The semicolon and all text following it until the next line, are ignored by the Lisp reader. It is considered good style to use three semicolons to indicate larger code blocks, two semicolons to preceed information about the code section following it, and one semicolon to add information about the code it is next to. Note that this is all convention, since one semicolon would be enough.[8]

In the large code fragment above we first defined some utility functions, then some general note transforming functions, to finally use these to define the dedicated ones that will be used in the coming examples. The utility functions for the pitch arithmetic isolate the calculation of pitches from the note transforming functions. This process of deciding for the abstraction layers is one of the essential aspects of programming style - something that is not demanded by the programming language itself but solely dependent on the taste and good judgement of the programmer.

Conditionals as control structure

With conditionals you can write functions that will, depending the value of one or more tests, evaluate a particular lisp expression. As an example take the function `maximum`.

```
(defun minimum (a b)

"Return minimum value of a and b"

(if (< a b)

a

b))
```

```
(minimum 6 4) -> 4
```

It takes two numbers as argument and returns the number with the smallest value. The function < (less-then) is a *predicate*, a function that returns a truth value, True or False (`t` or `nil` in Lisp). Furthermore, we use the special form `if`. It has as special syntax a test-form (here `(< a b)`) and two lisp expressions (here `a` and `b` respectively), the first one (the then-form) is evaluated and returned when the test-form is true, the second (the else-form) is evaluated and returned when it is false.

In Lisp the empty list `()` and the truth value False are defined equivalent and called `nil`. Conversly, everything that is not `nil` is considered as True: `t`. Other useful predicates are:

```
(equalp '(note 1 57 1) '(note 1 57 1)) -> t
```

```
(= 6 4) -> nil
```

```
(listp '(note 1 60 1)) -> t
```

```
(null '(note 1 60 1)) -> nil
```

The predicate `equalp` is the most general test for equality, `=` is specially for testing equality of numbers (although `equalp` would do as well)[9], `listp` tests whether its argument is a list, and `null` tests whether its argument is the empty list. [10]

Logical operators (`and`, `or` and `not`) are often useful in combining predicates.

```
(defun same-note? (note-1 note-2)
"Return true when notes are equal, false otherwise"
(and (= (duration note-1) (duration note-2))
(= (pitch note-1) (pitch note-2))
(= (loudness note-1) (loudness note-2))))
```

```
(defun note? (object)
"Return true if note, false otherwise"
(and (listp object)
```

```
(equalp (structural-type-of object) 'note)))


(same-note? '(note 1 61 1) '(note 1 61 0.7)) -> nil


(note? '(note 1 61 1)) -> t
```

There are different conditional constructs in Lisp. However, they can all be written in terms of the `if` construct. Take, for instance, the next two predicate functions using `if`:

```
(defun middle-c? (note)

"Return true if pitch of note is 60, false otherwise"

(if (= (pitch note) 60)

note

nil))


(defun sounding-note? (note)

"Return true if amplitude of note is non-zero, false otherwise"

(if (= (loudness note) 0)

nil

note))


(middle-c? '(note 1 61 1)) -> nil


(sounding-note? '(note 1 60 1)) -> (note 1 60 1)
```

Sometimes, to make the code more readable, one of the alternative conditionals are applicable. The two functions above, for example, can be written in a simpler form. When in an `if` there is no else-form, `when` should be used:

```
(defun middle-c? (note)

"Return true if pitch of note is 60, false otherwise"

(when (= (pitch note) 60) note))
```

When there is no then-form, `unless` should be used:

```
(defun sounding-note? (note)

"Return true if amplitude of note is non-zero, false otherwise"

(unless (= (loudness note) 0) note))
```

The body of `sounding- note?` could also have been written as `(when (not (= (loudness note) 0)) note)`, but this is considered less readable.

For more complicated conditionals we will use the more general construct `cond`. This construct has an elaborate syntax consisting of one or more clauses, each clause consisting of a test-form and one or more then-forms of which the last form is returned as result. Below, first a version using `if` is shown, followed by a more readable version using `cond`.

```
(defun amplitude-to-dynamic-marking (value)

"Return a dynamic marking associated with an amplitude value"

(if (< 0.8 value)

'ff

(if (< 0.6 value)

'f

(if (< 0.4 value)

'mp

(if (< 0.2 value)

'p

'pp)))))
```

```
(defun amplitude-to-dynamic-marking (value)

"Return a dynamic marking associated with an amplitude value"

(cond ((< 0.8 value) 'ff)

((< 0.6 value) 'f)

((< 0.4 value) 'mp)

((< 0.2 value) 'p)

(t 'pp)))
```

In the `cond` construct the test-form of the last clause, by convention, is always `T`. It contains the code for

the situation not covered by any of the other test-forms.[11] In the following function calls an amplitude value 0.66 is converted to the dynamic marking *fortissimo*, and the value 0.1 to *pianissimo*.

```
(amplitude-to-dynamic-marking 0.66) -> f
```

```
(amplitude-to-dynamic-marking 0.1) -> pp
```

The order of the clauses in a `cond` construct is important. Note that we actually use the order in which the test-forms are evaluated (i.e. starting from the top), using the knowledge that a certain test is only evaluated when previous tests have been false. In the function `amplitude- to- dynamic- marking` above using `cond` it is used to code that values between, for example, `0.6` and `0.8` are converted into `F`, by only testing whether the value is bigger than `0.6`. In some situations (e.g., with more complex tests) it is wiser to make this explicit, as shown in the next example. The predicate `between?` is introduced to abstract from the notion of a range:

```
(defun between? (value min max)
"Return true when value in interval <min, max], false otherwise"
(and (< min value) (<= value max)))

(defun amplitude-to-dynamic-marking (value)
"Return a dynamic marking associated with an amplitude value"
(cond ((between? value 0.8 1.0) 'ff)
((between? value 0.6 0.8) 'f)
((between? value 0.4 0.6) 'mp)
((between? value 0.2 0.4) 'p)
(t 'pp)))
```

The predicate <= tests whether its first argument is less than or equal to its second argument.[12]

We can also write the function that converts dynmamic markings to amplitudes using a `cond`. When the argument `mark` has none one of the values tested in the clauses, it will return `0.1`.[13]

```
(defun dynamic-marking-to-amplitude (mark)
"Return an amplitude value associated with a dynamic marking"
(cond ((equalp mark 'ff) 0.9)
```

```
((equalp mark 'f) 0.7)

((equalp mark 'mp) 0.5)

((equalp mark 'p) 0.3)

(t 0.1)))


(dynamic-marking-to-amplitude 'mp) -> 0.5
```

It is in these situations, where one form is tested to have different values the `case` contruct is appropiate. In the next example, the value of the argument is used as a key to select the appropiate amplitude associated with a dynamic mark:

```
(defun dynamic-marking-to-amplitude (mark)

"Return an amplitude value associated with a dynamic marking"

(case mark

(ff 0.9)

(f 0.7)

(mp 0.5)

(p 0.3)

(otherwise 0.1)))
```

The `case` contruct will evaluate its first argument (the key-form), then select a subsequent clause starting with that value (the key), followed by an evaluation the second part of that clause. The final case is preceded by `otherwise`, and contains the form to be evaluated when none of the other cases apply (like the `t` clause in `cond` construct).

Recursion as main control structure

Recursion provides an elegant way of reducing complex problems into simpler ones. This is done by first defining the solution to a simpler problem then recursively applying this solution (a piece of program code) to a more complex problem. A recursive function definition often has a characteristic pattern: a conditional expression, consisting of a stop-condition (or bottoming-out condition) for detecting the terminating non-recursive case of the problem, the result in that case (the stop-result), and the construction of a simpler, or reduced version of the overall problem by means of a recursive call. Below a template of a recursive function is shown. The aspects that still have to be filled in, dependent on the specific problem at hand, are bracketed with <>.

```
(defun <function name> <argument list>

(if <stop-condition>

<stop-result>
```

```
<recursive construction>))
```

As a first example, take the transposition of a collection, i.e. a list of notes.

```
(defun transpose-note-list-semitone (notes)
"Return a list of notes transposed by a semitone"
(if (null notes)
nil
(cons (transpose-note-semitone (first notes))
(transpose-note-list-semitone (rest notes)))))
```

When the list is empty (i.e. `(null notes)` is true) the task is simple: nothing has to be done. Otherwise we put (i.e. `cons`) the transposition of the first note in the result of transposing the `rest` of the list. Maybe surpisingly, the function required for transposing this smaller list is precisely the function we are writing at this moment, so it only has to call itself. This process of self-reference is called *recursion*. To improve readability, we could make the test-form of the `if` positive by replacing `(null notes)` by `notes`, i.e. testing whether there are notes, instead of testing the situation of no notes left:

```
(defun transpose-note-list-semitone (notes)
"Return a list of notes transposed by a semitone"
(if notes
(cons (transpose-note-semitone (first notes))
(transpose-note-list-semitone (rest notes)))
nil))
```

And, since the else-form is `nil`, we should use a `when` conditional:

```
(defun transpose-note-list-semitone (notes)
"Return a list of notes transposed by a semitone"
(when notes
(cons (transpose-note-semitone (first notes))
(transpose-note-list-semitone (rest notes)))))
```

In this final rewrite, the `when` condition tests whether there are any notes left. If so, it transposes the first one and puts it in the already transposed rest of the note list. If not, the result of the `when` clause will be

`nil`, and this will be the empty starting list for *consing* in transposed notes.

```
(transpose-note-list-semitone

'((note 1 60 1)(note 2 59 0.7)(note 1 65 0.7))) ->

((note 1 61 1)(note 2 60 0.7)(note 1 66 0.7))
```

Another example of a list recursive function is `integrate`. It can be used to convert a list of durations into a list of onsets by adding (starting from offset) the successive durations:

```
(defun integrate (numbers &optional (offset 0))

"Return a list of integrated numbers, starting from offset"

(if (null numbers)

(list offset)

(cons offset

(integrate (rest numbers)

(+ offset (first numbers))))))


(integrate '(1 2 3) 0) -> (0 1 3 6)
```

The function `integrate` uses the standard stop-condition for list recursion and constructs recursively a list of onsets by adding in every recursive call a duration to the current value of `offset`

Both `transpose- note- list- semitone` and `integrate` are examples of example of *list recursion.* They recur over a list, element by element, have a stop-condition that tests on lists and a recursive call where a new list is constructed. Another example of list recursion is shown in the next example:

```
(defun nth-element (n list)

"Return n-th element of list (zero-based)"

(cond ((null list)

nil)

((= n 0)

(first list))

(t (nth-element (- n 1) (rest list)))))
```

```
(nth-element 0 '(pp p mp f ff)) -> pp
```

```
(nth-element 2 '(pp p mp f ff)) -> mp
```

It is constructed using a slightly adapted recursive template, with an extra test added:

```
(defun <function name> <argument list>
(cond (<stop-condition>
<stop-result>)
(<specific-test>
<specific result>)
(t <recursive call>))
```

This extra test tests whether `n` is zero, and if so, returns the first elements of the list. In the final clause the function `nth- element` is called recursively with a decremented `n` and the rest of the list. Note that there is no new list constructed here. [14] We could use this function to rewrite, once again, `amplitude- to-dynamic- marking`:

```
(defun amplitude-to-dynamic-marking (value)
"Return a dynamic marking associated with an amplitude value"
(nth-element (round (* value 5)) '(pp p mp f ff)))
```

```
(amplitude-to-dynamic-marking .3) -> mp
```

While newly designed languages accepted recursion as control structure, Lisp was augmented with ‘down-to-earth’ and well-known iterative control structures, since it was recognized that in some cases it is simpler for humans to use instead of recursion (See chapter nil: imperative style). For complex cases however, the recursive form is often more elegant and easier to read. For example, if we wish to define a transposition on a complex musical structure (built from `parallel` and `sequential`), we must first dispatch on the type of structure (using a `case` construct) and then apply the transformation recursively on the component structures, to finally reassemble the transposed parts into their parallel or sequential order. The resultant program would look very messy when written iteratively. In general it can be said that recursion is the natural control structure for hierarchical data. And hierarchical structures are common in music.

```
(defun transpose-semitone (musical-object)
"Return a musical object transposed by a semitone"
```

```
(case (structural-type-of musical-object)

(note (transpose-note-semitone musical-object))

(sequential (make-sequential

(transpose-semitone (first-structure-of musical-object))

(transpose-semitone (second-structure-of musical-object))))

(parallel (make-parallel

(transpose-semitone (first-structure-of musical-object))

(transpose-semitone (second-structure-of musical-object))))))


(transpose-semitone (example)) ->

(sequential (parallel (note 1 61 1)

(note 2 64 1))

(note 4 56 0.5))
```

For *numeric recursion* we need to think of the appropiate the stop-condition and the recursive construction.

```
(defun <function name> <argument list>

(if <stop-condition>

<stop-result>

<recursive construction>))
```

The stop-condition will be a test on a number and the recursive construction will have to make new numbers (e.g., by multiplying them).

```
(defun exponent (number power)

"Return number raised to power"

(if (= power 0)

1

(* number (exponent number (- power 1)))))


(exponent 2 2) -> 4
```

The simplest case for `exponent` is a number raised to the power `0`, which is `1`. This is the stop condition in the function. Otherwise we multiply `number` by the result of `exponent` applied to `number` and the `power` decremented by one. We could use this function to convert from MIDI pitches to frequencies:[15]

```
(defun semitone-factor ()

"Return a constant factor"

1.0594630943592953)


(defun pitch-to-frequency (pitch)

"Translate a MIDI number into a frequency (in Hz.)"

(* 440.0 (exponent (semitone-factor) (- pitch 58))))


(pitch-to-frequency 70) -> 880.0
```

Finally, we return to list recursion with the function `retrogarde`, that reverses the elements in a list. We do this by puting the first note of the list at the end of the rest of the list, assuming that the rest is already reversed: we apply the function we are writing to it:

```
(defun retrogarde (notes)

"Return a reversed note list"

(when notes

(append (retrogarde (rest notes))

(list (first notes)))))


(retrogarde '((note 1 61 1)(note 2 64 1)(note 4 56 0.7))) ->

((note 4 56 0.7) (note 2 64 1) (note 1 61 1))
```

A more efficient way of coding this function is to use an *accumulating parameter*; instead of having to administrate all recursive calls to `retrogarde`, before being able to append one item to it, we can write the following[16]:

```
(defun retrogarde (notes)

"Return a reversed note list"

(accumulating-retrogarde notes nil))
```

```
(defun accumulating-retrogarde (notes result)

"Return a reversed note list, an auxiliary function"

(if (null notes)

result

(accumulating-retrogarde (rest notes)

(cons (first notes) result))))
```

This can be reduced to one function when we use an `&optional` argument. Note that `result` after the `&optional` is short for `(result nil)`.

```
(defun retrogarde (notes &optional result)

"Return a reversed note list"

(if (null notes)

result

(retrogarde (rest notes)

(cons (first notes) result))))
```

Initially, `retrogarde` is called with `result` being `nil`. [17] This parameter is then recursively filled with the elements of the note list and, when `notes` is empty, returned. The list will have all the notes in a reverse order because we put the first one in first, the second before that, etc.

In some situations, however, a simple *list iterator* would be enough. For instance, the function `mapcar` that applies a function to each item of a list and returns the individual results in a list. This function is useful for structures that have no hierarchy. For example, a flat list of notes like in the example below:

```
(mapcar #'pitch '((note 1 60 1)(note 1 62 1)(note 1 63 1))) -> (60 62 63)
```

[to do: ongeveer hier iets over een car/cdr recursie? En, symmetrisch, `mapcar` fun op elements and sub-elements, transpose uitschrijven?]

Binding as a way to retain intermediate results

In larger fragments of code it is often useful to temporary bind results that are used further on in the code. There are a number of binding constructs in Common Lisp that can be used for this The most common one is `let`, a special form, that thus comes with its own special syntax. It takes a list of variable bindings and then the code that can refer to these. The bindings are pairs of variable names and their values. In the following example the note `(note 1 60 1)` is bound to the variable name `note- a`, and `(note 2 62 .5)` to `note- b`. In the body these two values are combined into a sequential structure. They can be refered to

everywhere in the body of the `let`. However, outside the `let` they do not exist. This is called *lexical scope*: these variables can only be refered to within the establishing construct in which they are textually contained.

```
(let ((note-a '(note 1 60 1))

(note-b '(note 2 62 .5)))

(make-sequential note-a note-b)) ->

(sequential (note 1 60 1) (note 2 62 .5))
```

In a `let` all variables are calculated in parallel (i.e. first all values are calculated and then binded to variable names). When we want to use a variable-name in the description of a succeeding variable, we can use `let*`, that binds names sequentially:

```
(let* ((note-a '(note 1 60 1))

(note-b (make-note :duration 2

:pitch (pitch note-a)

:loudness .5)))

(make-sequential note-a note-b)) ->

(sequential (note 1 60 1) (note 2 60 .5))
```

Another useful binding construct is `destructuring- bind`. The first argument is a list with variable names. They are bound to the respective elements of the list that results from evaluating the expression, the second argument. Then the body of code follows, just like `let`.

```
(destructuring-bind (type note-a note-b)

'(sequential (note 1 60 1) (note 2 60 .5))

(make-sequential note-b note-a)) ->

(sequential (note 2 60 0.5) (note 1 60 1))
```

In fact, this is short-hand for writing:

```
(let* ((object '(sequential (note 1 60 1) (note 2 60 .5)))

(type (first object))

(note-a (second object))

(note-b (third object)))
```

```
(make-sequential note-b note-a))
```

`destructuring- bind` is especially useful when there are no accessors for the destructured data structure.

Functions as first class objects

In any good programming language all possible objects are allowed to appear in all possible constructs: they are all *first class citizens*. However in many programming languages this rule is often violated. In Lisp even exotic objects such as functions can be passed as an argument to a function (in an application construct) or yielded as a result from a function. At first sight this may seem unusual. PASCAL, for example allows the name of a procedure to be passed to another one using an ad hoc construction. And in C pointers to functions can be passed around. However, in Lisp all functions are uniformly considered as data objects in their own right, and functions operating on these them can be used. This provides an abstraction level that is a real necessity, but that is so often lacking in many other languages.

Functions as arguments

Suppose we want to write a function `mirror- around- middle- c` which would look similar to `transpose- semitone` defined before but only uses `mirror- note- around- middle- c` instead of `transpose- note- semitone` as the note transformation:

```
(defun mirror-around-middle-c (musical-object)

"Return a musical object mirror around middle c"

(case (structural-type-of musical-object)

(note (mirror-note-around-middle-c musical-object))

(sequential (make-sequential

(mirror-around-middle-c (first-structure-of musical-object))

(mirror-around-middle-c (second-structure-of musical-object))))

(parallel (make-parallel

(mirror-around-middle-c (first-structure-of musical-object))

(mirror-around-middle-c (second-structure-of musical-object))))))
```

Instead of just writing a new function for that purpose, it is better to abstract from the note transformation and write a general transform function. This function is now given the note transformation as an extra functional argument, which enables it to deal with all kinds of note transformations. Wherever it needs the result of the application of the note transformation function to a specific note. This is calculated with the Lisp `funcall` construct, that applies the value of its first argument (a function) to zero or more arguments:

```
(defun transform (musical-object note-transform)

"Return a transformed musical object"
```

```
(case (structural-type-of musical-object)

(note

(funcall note-transform musical-object))

(sequential

(make-sequential (transform (first-structure-of musical-object)

note-transform)

(transform (second-structure-of musical-object)

note-transform)))

(parallel

(make-parallel (transform (first-structure-of musical-object)

note-transform)

(transform (second-structure-of musical-object)

note-transform)))))
```

[to do: in (draw (transform ....)) voorbeelden, orginal-and-transform doen?]

We can now express both `mirror- note- around- middle- c` and `transpose- note- semitone` with one `transform` function:

```
(transform '(sequential (note 1 60 1)(note 2 63 1))

#'transpose-note-semitone) ->

(sequential (note 1 61 1)(note 2 64 1))
```

```
(transform '(sequential (note 1 60 1)(note 2 63 1))

#'mirror-note-around-middle-c) ->

(sequential (note 1 60 1)(note 2 57 1))
```

Note the use of the `#'` construct (called the *function quote*) which is used to signal to Lisp that the following expression is to be considered as a function. Next, we will define some more useful note-transformations:

```
(defun twice-note (note)

"Return a sequence of two notes"

(make-sequential note note))
```

```
(defun half-note (note)

"Return a note with half the duration"

(make-note :duration (/ (duration note) 2.0)

:pitch (pitch note)

:loudness (loudness note)))


(defun double-note (note)

"Return a doubled note"

(twice-note (half-note note)))


(double-note '(note 0.5 60 1)) ->

(sequential (note 0.25 60 1) (note 0.25 60 1))
```

The function `double- note` transformes a note into a sequence of two notes with half the original duration. It is built from two other transformations. The first one, `half- note`, divides the duration of its argument by two. The second, `twice- note`, makes a sequence of two identical copies of its argument. We can now use this function in combination with our `transform` function, doubling every note in the musical object.

```
(transform (example) #'double-note) ->

(sequential (parallel (sequential (note 0.5 60 1)(note 0.5 60 1))

(sequential (note 1 63 1)(note 1 63 1)))

(sequential (note 2 55 0.5)(note 2 55 0.5)))


(draw (transform (example) #'double-note)) =>
```

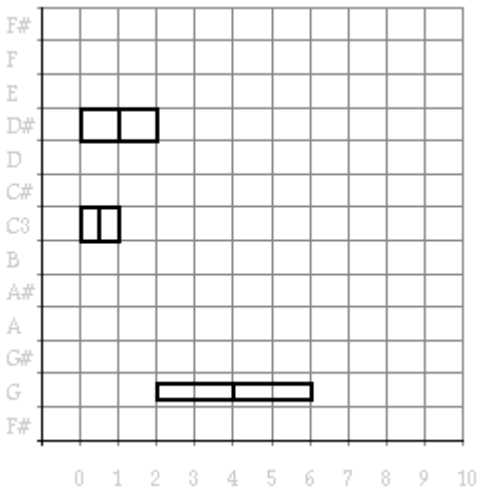

Figure 4. Example transformed with double-note

The use of functions as arguments (so called *downward funargs*) seems to give so much extra power that we might begin to wonder what good the passing of functions as results (so called *upward funargs*) could give us.

Functions as results

If we wanted to apply an octave transposition to a structure we would have to write a new function, `transpose- note- octave`, and use it as an argument for `transform`.

```
(defun transpose-note-octave (note)

"Return a note transposed by an octave"

(transpose-note note 12))
```

```
(transform (example) #'transpose-note-octave) ->

(sequential (parallel (note 1 72 1)

(note 2 75 1))

(note 4 67 0.5))
```

This means we always have to define the possible transformations in advance. This is not satisfactory and instead we could use *anonymous functions* as an argument to the transform function. Anonymous functions are not as bad as they look. They are merely a consequence of the rule of first class citizens. For example, it is perfectly normal for objects like numbers, lists and strings to have a notation for the constant values (you can write `(+ 1 1)` if you need to notate the constant `2`). Functions should also have this property. The anonymous function of one argument (`note`), that will transpose a note by an octave, can be notated like this:

```
#'(lambda (note) (transpose-note note 12))
```

```
(funcall #'(lambda (note) (transpose-note note 12)) '(note 1 60 1)) ->

(note 1 72 1)
```

This kind of function can be used as argument to the `transpose` function (remember the function-quote):

```
(transform (example) #'(lambda (note) (transpose-note note 12))) ->

(sequential (parallel (note 1 72 1) (note 2 75 1)) (note 4 67 0.5))


(transform (example)

#'(lambda (note) (mirror-note note 60))) ->

(sequential (parallel (note 1 60 1) (note 2 57 1)) (note 4 65 0.5))
```

Still this is a little tedious to do, we have to construct a new anonymous function for every transposition interval or mirroring pitch. We can, however, with the tools we have now, define the functions that will calculate these transposition and mirror functions when given the appropiate argument: functions that construct functions according to our definitions (*program generators*). This is a very powerful mechanism.

```
(defun transpose-note-transform (interval)

"Return a function for transposing a note by interval"

#'(lambda (note) (transpose-note note interval)))


(defun mirror-note-transform (center)

"Return a function for mirroring a note around a center"

#'(lambda (note) (mirror-note note center)))


(draw (transform (example) (transpose-note-transform 2))) =>
```

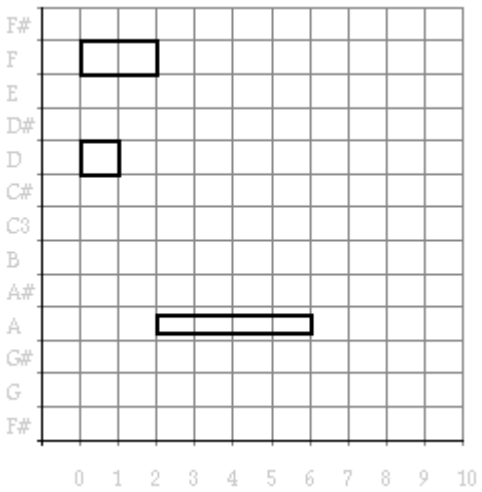

Figure 5. Example transformed with transpose-note-transform

```
(draw (transform (example) (mirror-note-transform 59))) =>
```

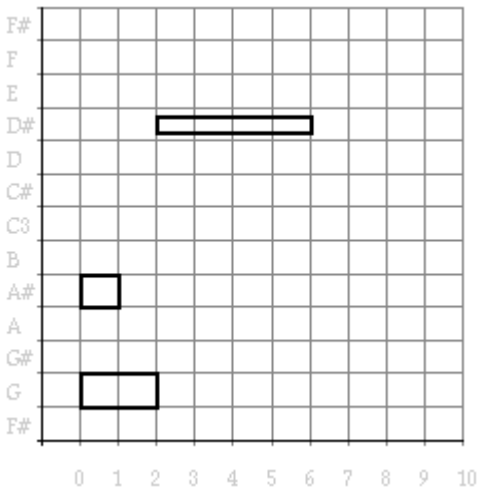

Figure 6. Example transformed with mirror-note-transform

When we add an extra `&rest` argument `args`, our transform can also be used with note transformations of more than one argument:

```
(defun transform (musical-object note-transform &rest args)
"Return a transformed musical object"
(case (structural-type-of musical-object)
(note
(apply note-transform musical-object args))
(sequential
(make-sequential (apply #'transform
(first-structure-of musical-object)
note-transform
args)
(apply #'transform
(second-structure-of musical-object)
note-transform
args)))
```

```
(parallel

(make-parallel (apply #'transform

(first-structure-of musical-object)

note-transform

args)

(apply #'transform

(second-structure-of musical-object)

note-transform

args)))))
```

```
(draw (transform (example) #'limit-loudness 0.5 0.8)) =>
```

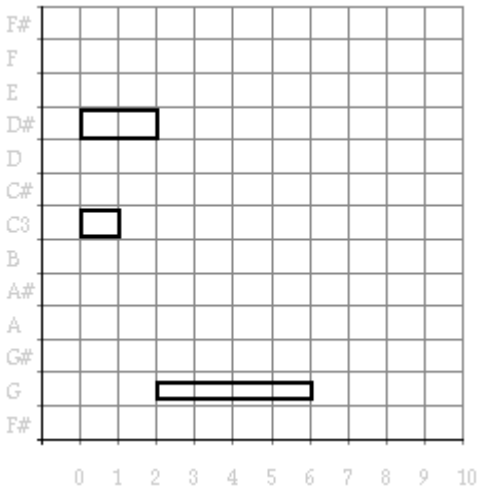

Figure 7. Loudness transformed example

```
(draw (transform (example) #'transpose-note 3)) =>
```

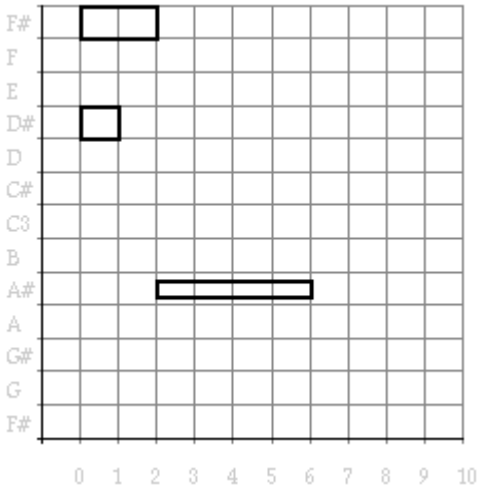

Figure 8. Pitch transformed example

Note that we had to change the `funcall` of the note-transform into an `apply` and add an `apply` before every call to `transform`. `Apply` is needed when there a list of arguments, each of which should be individually communicated to the applied function. The last argument to apply should be a list. Compare their use in the following examples:

```
(funcall #'make-sequential '(note 1 60 1) '(note 1 62 1)) ->

(sequential (note 1 60 1) (note 1 62 1) )
```

```
(apply #'make-sequential '((note 1 60 1)(note 1 62 1))) ->

(sequential (note 1 60 1) (note 1 62 1))
```

```
(apply #'make-sequential '(note 1 60 1) '((note 1 62 1))) ->

(sequential (note 1 60 1) (note 1 62 1))
```

[to do: iets over `eval, apply, &rest` interactie hier]

Generality as aim

Functional programming makes it possible to construct very general programs that are customized for specific purposes. These are the tools that are badly needed in software design. They deserve to be supplied in software libraries, so that programmers can stop reinventing the wheel each time. As a tool for composers these programs may aim to be as ‘empty’ as possible, only expressing general abstract knowledge about musical structure and leaving open details about the specific material and relations used. The transformations we have so far designed are not yet that general. They were structure preserving, and

thus a transformation of a sequence would always yield a sequence. Only at the note level could the structure expand into a bigger one (e.g., when using `double- note`). Bearing this in mind we are going to develop a more general transformation device.

Our new transformation is like the old one, except that it takes two more arguments to calculate what a `sequential`, and what a `parallel` structure transforms into:

```
(defun transform (musical-object sequential-transform
parallel-transform
note-transform
&rest args)
"Return a transformed musical object"
(case (structural-type-of musical-object)
(note
(apply note-transform musical-object args))
(sequential
(funcall sequential-transform
(apply #'transform
(first-structure-of musical-object)
sequential-transform
parallel-transform
note-transform
args)
(apply #'transform
(second-structure-of musical-object)
sequential-transform
parallel-transform
note-transform
args)))
(parallel
(funcall parallel-transform
(apply #'transform
(first-structure-of musical-object)
sequential-transform
```

```
parallel-transform

note-transform

args)

(apply #'transform

(second-structure-of musical-object)

sequential-transform

parallel-transform

note-transform

args)))))
```

We now have available a very powerful transformation device. First a rather stupid example of its use: the identity (i.e. no tranformation) transformation which just rebuilds its arguments.

```
(defun no-note-transform (note)

"Return untransformed note"

note)


(transform (example) #'make-sequential #'make-parallel #'no-note-transform) ->

(sequential (parallel (note 1 60 1)

(note 2 63 1))

(note 4 55 0.5))
```

In Figure 9 is shown how the results (i.e. musical structures) are passed upward by `no- note-transform`, `make- parallel` and `make- sequential` .

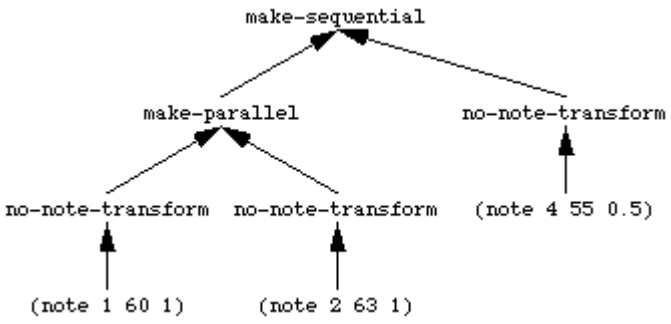

Figure 9. Evaluation hierarchy.

[to do: these trees should be eval-hierarchies]

Now some more useful transformations can be constructed. The results that are passed upward are numbers. The first transformation calculates the duration of a complex musical structure by adding or maximizing the duration of substructures: adding durations for `sequential` structures, and taking the maximum of each element for `parallel` structures:

```
(defun duration-of (musical-object)
"Return duration of musical object"
(transform musical-object #'+ #'max #'duration))
```

Similarly it is possible to calculate the duration of the longest note, the number of notes in a piece and the maximum number of parallel voices of a complex structure. Note how easy the `transform` function is adapted to these different purposes by 'plugging in' different functional arguments.

```
(defun longest-note-duration (musical-object)
"Return longest-note-duration in musical object"
(transform musical-object #'max #'max #'duration))

(defun count-one (note)
"Return the number 1"
1)

(defun number-of-notes (musical-object)
"Return the number of notes in musical object"
(transform musical-object #'+ #'+ #'count-one))

(defun max-number-of-parallel-voices (musical-object)
"Return the maximum number of parallel voices in musical object"
(transform musical-object #'max #'+ #'count-one))

(duration-of (example)) -> 6
```

```
(longest-note-duration (example)) -> 4
```

```
(number-of-notes (example)) -> 3
```

```
(max-number-of-parallel-voices (example)) -> 2
```

Figures 10to 13show the way information is communicated upward in the different uses of `transform`.

[to do: these trees should be eval-hierarchies]

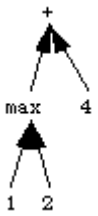

Figure 10. (duration-of (example)) -> 6

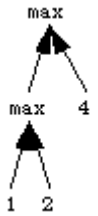

Figure 11. (longest-note-duration (example)) -> 4

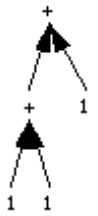

Figure 12. (number-of-notes (example)) -> 3

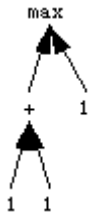

Figure 13. (max-number-of-parallel-voices (example)) -> 2

To demonstrate again the generality of the transform, we will now write a program to draw a piano-roll notation of a musical structure as shown in previous figures. To draw a note at the correct place we need to know the absolute start time of a musical object, information that neither the notes or the transform function itself supplies. When context information is missing, it is a well known trick in AI programming, to calculate a function of the (not yet known) context as temporary result. We could use such a solution here. `draw- note` function returns a function that will draw a graphical representation of a note when given a start time. As the drawing is done as a *side- effect*, this function can then return the end time of the note as context (start-point) to use in further drawing. `draw- sequential` function receives two such draw-functions as arguments and constructs the draw function that will pass its start time to the first and pass the end time returned by the first to the second, returning its end time as the result. function `draw-parallel` will pass its start time to both sub-structure draw functions returning the maximum end time they return. Thus not numbers or musical structures are passed upward as result of the transformation on sub-structures, but functions that can draw the sub-structure when given a start time. function `draw-musical- object` just has to apply the draw function resulting from the call to transform to time 0. function `draw` is the top level function that constructs a graphical window using `make- draw- window` (that we will be described in section 3.1: object oriented I) and applied our `draw- musical- object` to it.

```
(defun draw (musical-object &rest args)

"Make a window and draw the musical object on it"

(let ((window (apply #'make-piano-roll-window args)))

(draw-musical-object musical-object window)

window))


(defun draw-musical-object (musical-object window)

"Draw a musical object on window"

(funcall (transform musical-object

#'draw-sequential

#'draw-parallel
```

```
#'draw-note
window)
0))

(defun draw-note (note window)
"Return end time of note, drawn as side effect"
#'(lambda (time)
(draw-horizontal-block time ; x-position
(pitch note) ; y-position
(duration note) ; width
(loudness note) ; height
window) ; output window
(+ time (duration note)) ; end time
))

(defun draw-sequential (a b)
"Return end time of sequential object"
#'(lambda (time)
(funcall b (funcall a time))))

(defun draw-parallel (a b)
"Return end time of parallel object"
#'(lambda (time)
(max (funcall b time)
(funcall a time))))

(defun draw-horizontal-block (left middle width height window)
"Draw block on window"
(let* ((right (+ left width))
(top (+ middle (* height .5)))
(bottom (- middle (* height .5)))
(boundary (list left top right bottom)))
(draw-rectangle window boundary) ; assumed graphical primitive
window))
```

```
(draw (example)) =>
```

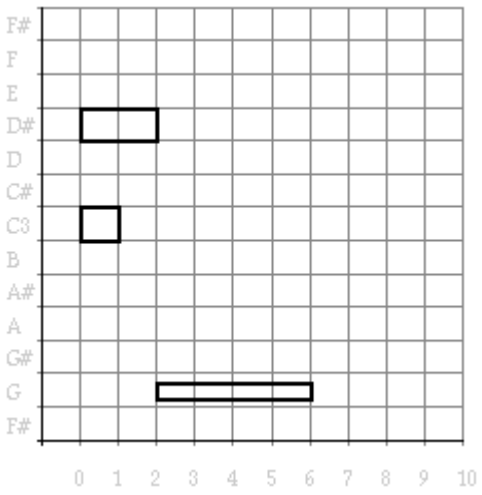

Figure 14. The piano roll representation of the example.

Combinators as function builders

Since it turned out to be so useful to be able to talk about functions as objects which are passed to and from other functions, we are now going to examine the possibilities of a special kind of these “other” functions, called *combinators*. A combinator is a higher order function that has only functions as arguments and return s a function as a result. The first one we will show is the combinator called `twice`:

```
(defun twice (transform)
"Return a function that applies a transform twice"
#'(lambda(object)
(funcall transform (funcall transform object))))

(funcall (twice #'rest) '(1 2 3 4 5)) -> (3 4 5)
```

It can double the action of any function. and therefore `(twice #'rest)` will be a function that removes the first two elements from a list. `(twice #'transpose- note- octave)` will yield a function that transposes notes two octaves and `(twice #'mirror- note- around- middle- c)` will be a useless transformation (the identical transformation).

```
(transform (example)
```

```
#'make-sequential

#'make-parallel

(twice #'transpose-note-octave)) ->

(sequential (parallel (note 1 84 1)

(note 2 87 1))

(note 4 79 .5))
```

The second combinator is the “function-composition” combinator. It can combine the actions of two transformations into a new one.[18].

```
(defun compose (transform-1 transform-2)

"Return function composes of two other functions"

#'(lambda(object)

(funcall transform-1 (funcall transform-2 object))))


(funcall (compose #'first #'rest) '(c d e f g a b)) -> d
```

To construct a transformation that is a doubling applied to the result of an octave transposition we could use this combinator to build it.

```
(transform (example)

#'make-sequential

#'make-parallel

(compose #'double-note #'transpose-note-octave)) ->

(sequential

(parallel (sequential (note .5 72 1)(note .5 72 1))

(sequential (note 1 75 1)(note 1 75 1)))

(sequential (note 2 67 .5)(note 2 67 .5)))


(draw (transform (example)

#'make-sequential

#'make-parallel

(compose #'double-note #'transpose-note-octave))) =>
```

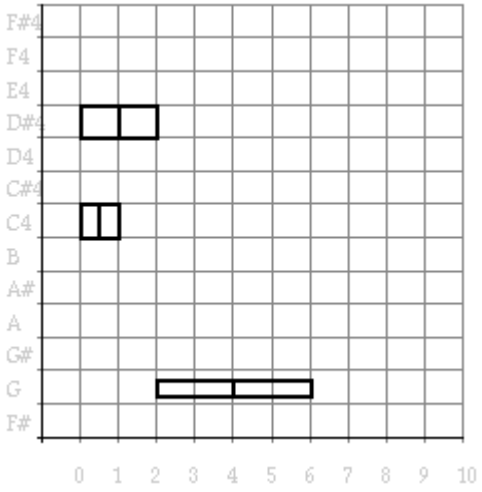

Figure 15. The piano roll representation of doubled and transposed notes.

Note that in Figure 15 the y-axis is adapted.

To show the usefulness of these constructions, we will write a function that calculates a complex melody from a simple one by adding a parallel melody that is the doubling of the original one transposed one octave. This is a transformation often used in Javanese Gamelan music. The score of the `add- doubled` transformation on the `example` object is shown in Figure 16.

```
(defun add-doubled (musical-object)

"Return a musical-object with itself doubled and octave higher added"

(make-parallel musical-object

(transform musical-object

#'make-sequential

#'make-parallel

(compose #'double-note

#'transpose-note-octave))))


(draw (transform (example) #'make-sequential #'make-parallel #'add-doubled)) =>
```

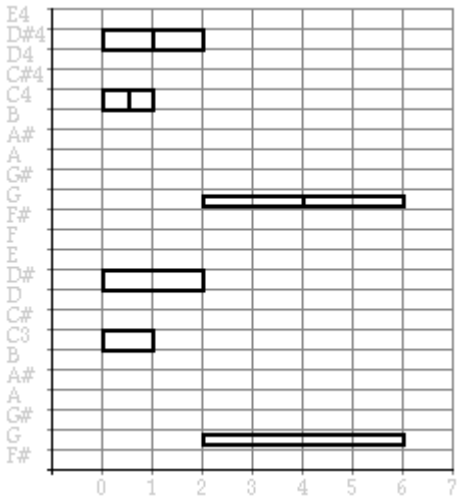

Figure 16. The piano roll representation of add-doubled transformation.

Note that we could have defined `twice` as a composition of a transform with itself:

```
(defun twice (transform)
"Return a function that applies a transform twice"
(compose transform transform))
```

Next we will write some logical combinators that take predicte as argument and return a new one. A simple `compose- and- 2`, composing two predicates would look like this:

```
(defun compose-and-2 (predicate-a predicate-b)
"Return predicate that is a logical and of predicates"
#'(lambda(&rest args)
(and (apply predicate-a args)
(apply predicate-b args))))

(funcall (compose-and-2 #'note?
#'middle-c?)
'(note 1 60 1)) ->
(note 1 60 1)
```

We use a `&rest` lambda-list keyword and an `apply` to make it work for predictates that take more than one argument.

Next we can write a `compose- and` and `compose- or` that can take more than two predicates as follows:

```
(defun compose-and (predicate &rest predicates)
"Return predicate that is a logical and of predicates"
#'(lambda(&rest args)
(and (apply predicate args)
(apply (apply #'compose-and predicates) args))))

(defun compose-or (predicate &rest predicates)
"Return predicate that is a logical or of predicates"
#'(lambda(&rest args)
(or (apply predicate args)
(apply (apply #'compose-or predicates) args))))
```

Below a predicate that tests whether the argument is a sounding middle-c:

```
(funcall (compose-and #'note?
#'sounding-note?
#'middle-c?)
'(note 1 60 0)) ->
nil
```

More on generality as aim

We will now return to our `transform` function. We saw that it was a general solution for dealing with context information. However, this will not be always the best solution. When information like start-time is used a lot, it may be wiser to adapt the transform function itself so that it passes this information as well to the note-transformation function. Instead of rewriting `transform` itself, we will wrap a function of time around our sequential, parallel en note-transform arguments, such that they can be used in our existing transform function.

```
(defun timed-transform (musical-object time
sequential-transform
```

```
parallel-transform

note-transform

&rest args)

"Return a transformed musical object"

(first (funcall (apply #'transform

musical-object

(wrap-sequential sequential-transform)

(wrap-parallel parallel-transform)

(wrap-note note-transform)

args)

time)))
```

We will start by writing the note wrapper. `wrap- note` will return a function of one argument (the note object), but will apply the transform to both the note and time:

```
(defun wrap-note (transform)

"Return a note transform that has access to its start-time"

#'(lambda(note &rest args)

#'(lambda(time)

(apply transform note time args))))
```

Next we rewrite it to return both the transformed note and the end time. In the `draw` code, where we used this technique for the first time, we could just return the end times of notes (to be used, e.g., to calculate the start time of the next note) because drawing was done as a side-effect. Here we do need the transformed objects as well, so we return both values, the transformed note and its end time, in a list:

```
(defun wrap-note (transform)

"Return a note transform that has access to its start-time"

#'(lambda(note &rest args)

#'(lambda(time)

(list (apply transform note time args)

(+ time (duration note))))))
```

The seguential and parallel wrappers will collect these values, apply their transform, and communicate the

new objects and end times:

```
(defun wrap-sequential (transform)
"Return a sequential transform that has access to its start-time"
#'(lambda(object-a object-b)
#'(lambda(time)
(destructuring-bind (result-a end-a) (funcall object-a time)
(destructuring-bind (result-b end-b) (funcall object-b end-a)
(list (funcall transform result-a result-b)
end-b))))))

(defun wrap-parallel (transform)
"Return a sequential transform that has access to its start-time"
#'(lambda(object-a object-b)
#'(lambda(time)
(destructuring-bind (result-a end-a) (funcall object-a time)
(destructuring-bind (result-b end-b) (funcall object-b time)
(list (funcall transform result-a result-b)
(max end-a end-b)))))))
```

The function `wrap- sequential` applies a wrapped `object- a` (a `note`, `sequential` or `parallel` object) to the current time, collects its result and end time, and then applies the other wrapped obect (`object- b`) to this end time. Next the `transform` function is applied to both results and returned with the end time of the last object in the sequence. The function `wrap- parallel` does a similar thing, however apllied both wrapped objects to the same time and returns the maximim of both end times. This is advanced use of functions, so take your time in understanding it.

With our new `timed- tranform` our previous piano-roll drawing code becomes obsolete: we just need to communicate a draw-function for notes, and do nothing otherwise:

```
(defun nothing (&rest ignore)
"Return nil and do nothing"
nil)

(defun draw-musical-object (musical-object window)
```

```
"Draw a musical object on window"

(timed-transform musical-object

0

#'nothing

#'nothing

#'(lambda(note time window)

(draw-horizontal-block time

(pitch note)

(duration note)

(loudness note)

window))

window))


(draw (example)) =>
```

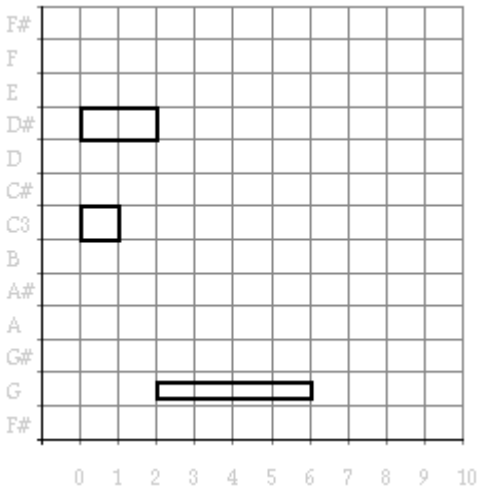

Figure 17. Piano-roll example using timed-transform.

We now can built transformations, such as a `fade- out` (decrescendo), that are time dependent. For that, we first will make a *time function*. The function `decline- time- function` is a constructor function: it returns a function of time. It will return values from 1 to 0 (linearly interpolated) over the interval defined by `begin` and `end`.

```
(defun decline-time-function (begin end)
```

```
"Return a time-function, a ramp from 1 to 0 over interval [begin, end]"

#'(lambda (time)

(- 1.0

(/ (- time begin)

(- end begin)))))


(funcall (decline-time-function 0 10) 0) -> 1.0


(funcall (decline-time-function 0 10) 2) -> 0.8
```

We use this function to write a `fade- out- transform` for notes, scaling the loudness of notes with this time-function.

```
(defun fade-out-transform (begin end)

"Return note with decresenco applied to it"

#'(lambda (note time)

(if (< begin time end)

(make-note :duration (duration note)

:pitch (pitch note)

:loudness (* (loudness note)

(funcall (decline-time-function begin end) time)))

note)))
```

`fade- out- transform` returns a function of two arguments. This function, when applied to a note at a certain time, will return a new note with a loudness depending on the position within the time interval delimited by `begin` and `end`.

```
(draw (timed-transform (make-sequential (example) (example))

0

#'make-sequential

#'make-parallel

(fade-out-transform 0 11))) =>
```

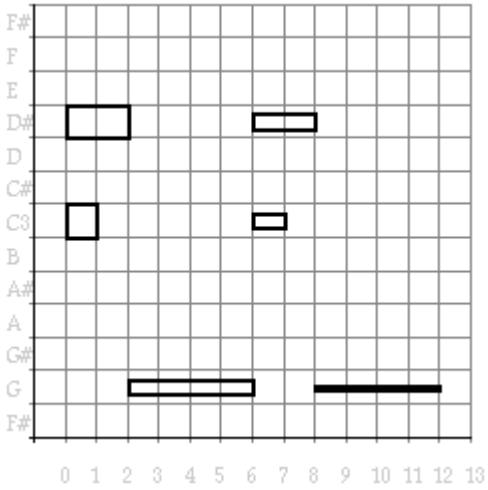

Figure 18. fade-out example

Sometimes we wish to transform our musical objects to note lists where we add the start time of each note. For example, when we want to play our musical object. All that is required is a function to transform a note to a data structure containing the note and its start time: a timed note. We add a new contructor and two accessors:

```
(defun make-timed-note (note onset)

"Return a note with start time"

(list (list onset note)))


(defun start-of (timed-note)

"Return start-time of timed-note data structure"

(first timed-note))


(defun note-of (timed-note)

"Return note of timed-note data structure"

(second timed-note))
```

We can now write a function `musical- object- to- note- list` that transforms one of our musical objects into a flat list of notes, using our new `timed- tranform`. In the case of notes we add the start-time to it (with `make- timed- note`) and add . For sequential objects we simply `append` its elements, since they are already in the right time order. Only for parallel objects we have to do a little more work. Its elements, which will be two note lists in the right order, will have to be merged. For this we write the function `merge- note- lists` which takes two note-lists and, depending on the onset of the notes, conses

the first note of one list or the first note of the other in the resulting merged list, and this of course recursively. Note the more elaborate list-recursion template used here, with two stop-conditions, an extra test, and two recursive constructions.

```
(defun merge-note-lists (list-1 list-2)

"Return merged notes list"

(cond ((null list-1) list-2)

((null list-2) list-1)

((<= (start-of (first list-1))

(start-of (first list-2)))

(cons (first list-1)

(merge-note-lists (rest list-1) list-2)))

(t (cons (first list-2)

(merge-note-lists list-1 (rest list-2))))))


(defun musical-object-to-note-list (musical-object)

"Return a flat note-list derived from musical object"

(timed-transform musical-object

0

#'append

#'merge-note-lists

#'make-timed-note))


(musical-object-to-note-list (example)) ->

((0 (note 1 60 1)) (0 (note 2 63 1)) (2 (note 4 55 0.5)))
```

Using this function we can play the example (readers we will just to have to look at it).

```
(defun play (musical-object)

"Play a musical object"

(play-notes (musical-object-to-note-list musical-object)))
```

Figure 19. Score of the played example.

Finally, we will improve of our constructors `make- sequential` and `make- parallel` such that they are not restricted to two arguments. We can use the lambda-list keyword `&rest` that signals Lisp to collect all the arguments of the function in a list:

```
(defun make-sequential (&rest elements)
"Return a sequential data structure"
(cons 'sequential elements))

(defun make-parallel (&rest elements)
"Return a parallel data structure"
(cons 'parallel elements))

(defun elements (structured-object)
"Return elements of object"
(rest structured-object))

(make-sequential (make-note :duration 1 :pitch 60)
(make-note :duration 1 :pitch 62)
(make-note :duration 1 :pitch 63)) ->
(sequential (note 1 60 1)(note 1 62 1)(note 1 63 1))

(elements '(sequential (note 1 60 1)(note 1 62 1)(note 1 63 1))) ->
((note 1 60 1)(note 1 62 1)(note 1 63 1))
```

We have to adapt `transform` to deal with these new definitions, using `mapcar` to iterate the transform over all the elements of a `sequential` or `parallel` structure:

```
(defun transform (musical-object sequential-transform
parallel-transform
note-transform
&rest args)
"Return a transformed musical object"
```

```
(case (structural-type-of musical-object)
(note (apply note-transform musical-object args))
(sequential
(apply sequential-transform
(mapcar #'(lambda(element)
(apply #'transform
element
sequential-transform
parallel-transform
note-transform
args))
(elements musical-object))))
(parallel
(apply parallel-transform
(mapcar #'(lambda(element)
(apply #'transform
element
sequential-transform
parallel-transform
note-transform
args))
(elements musical-object))))))

(defun timed-transform (musical-object time
sequential-transform
parallel-transform
note-transform
&rest args)
"Return a transformed musical object"
(first (funcall (apply #'transform
musical-object
(wrap-sequential sequential-transform)
(wrap-parallel parallel-transform)
(wrap-note note-transform)
```

```
args)

time)))
```

The wrappers for our `timed- transform`, that uses our newly defined `transform` (code repeated above), can be simplified a lot using a sequential and parallel iterator that communicate the approiate end-time to the transformation functions:

```
;;; iterators


(defun sequential-iterator (funs time &optional results)

"Return result of applying transformations sequentically"

(if (null funs)

(list (reverse results) time)

(destructuring-bind (result end) (funcall (first funs) time)

(sequential-iterator (rest funs) end (cons result results)))))


(defun parallel-iterator (funs time &optional results (max time))

"Return result of applying transformations in parall"

(if (null funs)

(list (reverse results) max)

(destructuring-bind (result end) (funcall (first funs) time)

(parallel-iterator (rest funs) time (cons result results) (max end time)))))


;;; wrappers


(defun wrap-sequential (transform)

"Return a sequential transform that has access to its start-time"

#'(lambda(&rest wrapped-objects)

#'(lambda(time)

(destructuring-bind (results end)

(sequential-iterator wrapped-objects time)

(list (apply transform results) end)))))


(defun wrap-parallel (transform)
```

```
"Return a parallel transform that has access to its start-time"
#'(lambda(&rest wrapped-objects)
#'(lambda(time)
(destructuring-bind (results end)
(parallel-iterator wrapped-objects time)
(list (apply transform results) end)))))

(defun wrap-note (transform)
"Return a note transform that has access to its start-time"
#'(lambda(note &rest args)
#'(lambda(time)
(list (apply transform note time args)
(+ time (duration note))))))

(defun large-example ()
"Return large musical object"
(make-sequential (example) (example) (example)))

(draw (timed-transform (large-example)
0
#'make-sequential
#'make-parallel
(fade-out-transform 0 18))) =>
```

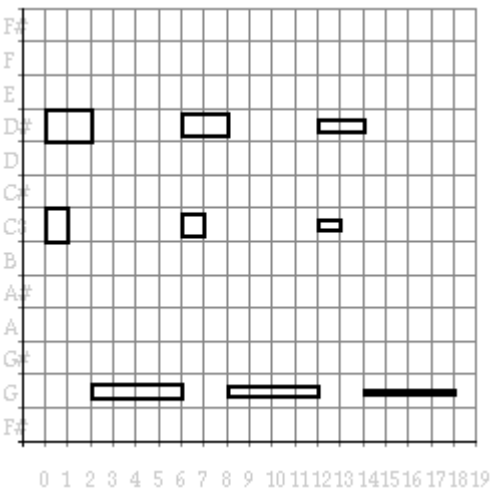

Figure 20. Piano-roll of larger example.

Parameters as superfluous

`defun` can be considered as a device that makes the definition of a function easier to read, but it assumes the name of the function and the function body itself to be constants. `define- function` is a similar construct that gives a little bit more power than `defun` that we can use for calculation functions. We will make this construct in chapter nil: imperative styleor (CHAPTER-REF “embedded style”)??, for now we will just use it. It takes a name, an anonymous function, and a documentation string. The following expression (re)defines the function `duration`

```
(define-function duration

#'(lambda(note) (second-element note))

"Return the duration of note")
```

With this construct we can calculate a function body instead of using a constant anonymous function. In the example below we calculate one using one of the transformations defined before.

```
(define-function transpose-note-octave

(transpose-note-transform 12)

"Return a note transposed by an octave")


(transpose-note-octave '(note 1 60 1)) -> (note 1 72 1)
```

Note that in the definitions above the formal parameters, which appeared in the argument list of the `defun` form, are no longer needed. If we have access to enough combinators like `twice` and `compose`, we can even completely do without parameters.

```
(define-function double-note

(compose #'twice-note #'half-note)

"Return a doubled note")


(define-function double-and-raise-note

(compose #'double-note #'transpose-note-octave)

"Return a doubled note one octave higher")
```

```
(double-and-raise-note '(note 1 60 1)) ->

(sequential (note 0.5 72 1) (note 0.5 72 1))
```

Languages built on combinators using *parameter-freeprogramming* are very useful in domains centered around one type of object and many transformations on this object (like the musical structures in our examples). In these domains they facilitate the definition of higher levels of abstraction whereby transformations are considered objects in their own right, so that they can be manipulated, combined and modified. However, when dealing with functions of many arguments we need a lot of combinators for juggling with the order of arguments leading to programs that are difficult to read. For humans, an extra hook into our memory, by means of a mnemonic parameter name, is often indispensable. In addition more heterogeneous domains consisting of different sorts of objects, all subjected to transformations that are conceptually more or less the same, can be modelled better using another style of programming in Lisp. In this style named *Object Oriented Programming* it is straightforward to express, for example, how both a melody and a synthesizer can have their own definition of `transposition`. This is the subject of Chapter chapter nil: Object oriented. In the next chapter (chapter nil: imperative style) we will elaborate the use of program generators using `define- function`.

Definitions made

`draw-musical-object (musical-object window)` *`function`*

a musical object on window

`draw-horizontal-block (left middle width height window)` *`function`*

block on window

`draw (musical-object &rest args)` *`function`*

a window and draw the musical object on it

`play (musical-object)` *`function`*

a musical object

`semitone-factor nil` *`function`*

a constant factor

`example nil` *`function`*

a constant musical object

`double-note (note)` *`function`*

a doubled note

`amplitude-to-dynamic-marking (value)` *`function`*

a dynamic marking associated with an amplitude value

`musical-object-to-note-list (musical-object)` *`function`*

a flat note-list derived from musical object

`mirror-note-transform (center)` *`function`*

a function for mirroring a note around a center

`transpose-note-transform (interval)` *function*

a function for transposing a note by interval

`twice (transform)` *function*

a function that applies a transform twice

`integrate (numbers &optional (offset 0))` *function*

a list of integrated numbers, starting from offset

`transpose-note-list-semitone (notes)` *function*

a list of notes transposed by a semitone

`mirror-around-middle-c (musical-object)` *function*

a musical object mirror around middle c

`transpose-semitone (musical-object)` *function*

a musical object transposed by a semitone

`add-doubled (musical-object)` *function*

a musical-object with itself doubled and octave higher added

`make-note (&key (duration 1) (pitch 60) (loudness 1))` *function*

a note data structure

`wrap-note (transform)` *function*

a note transform that has access to its start-time

`transpose-note-octave (note)` *function*

a note transposed by an octave

`half-note (note)` *function*

a note with half the duration

`make-timed-note (note onset)` *function*

a note with start time

`make-parallel (&rest elements)` *function*

a parallel data structure

`wrap-parallel (transform)` *function*

a parallel transform that has access to its start-time

`retrogarde (notes &optional result)` *function*

a reversed note list

`accumulating-retrogarde (notes result)` *function*

a reversed note list, an auxiliary function

`twice-note (note)` *function*

a sequence of two notes

`make-sequential (&rest elements)` *function*

a sequential data structure

`wrap-sequential (transform)` *function*

a sequential transform that has access to its start-time

`decline-time-function (begin end)` *function*

a time-function, a ramp from 1 to 0 over interval [begin, end]

`timed-transform (musical-object time sequential-transform parallel-transform note-transform &rest args)` *function*

a transformed musical object

`transform (musical-object sequential-transform parallel-transform note-transform &rest args)` *function*

a transformed musical object

`structural-type-of (musical-object)` *function*

a type of a musical object

`dynamic-marking-to-amplitude (mark)` *function*

an amplitude value associated with a dynamic marking

`duration-of (musical-object)` *function*

duration of musical object

`elements (structured-object)` *function*

elements of object

`draw-note (note window)` *function*

end time of note, drawn as side effect

`draw-parallel (a b)` *function*

end time of parallel object

`draw-sequential (a b)` *function*

end time of sequential object

`fourth-element (list)` *function*

fourth element of list

`compose (transform-1 transform-2)` *function*

function composes of two other functions

`large-example nil` *function*

large musical object

`longest-note-duration (musical-object)` *function*

longest-note-duration in musical object

`merge-note-lists (list-1 list-2)` *function*

merged notes list

`minimum (a b)` *function*

minimum value of a and b

`nth-element (n list)` *function*

n-th element of list (zero-based)

`nothing (&rest ignore)` *function*

nil and do nothing

`mirror-note (note center)` *function*

note mirrored around center

`mirror-note-around-middle-c (note)` *function*

note mirrored around middle-c

`note-of (timed-note)` *function*

note of timed-note data structure

`transpose-note (note interval)` *function*

note transposed by interval

`transpose-note-semitone (note)` *function*

note transposed by semi-tone

`limit-loudness (note low high)` *function*

note with clipped loudness

`fade-out-transform (begin end)` *function*

note with decresenco applied to it

`exponent (number power)` *function*

number raised to power

`transpose-pitch (pitch interval)` *function*

pitch increased by interval

`mirror-pitch (pitch center)` *function*

pitch mirrored around center

`compose-and (predicate &rest predicates)` *function*

predicate that is a logical and of predicates

`compose-and-2 (predicate-a predicate-b)` *function*

predicate that is a logical and of predicates

`compose-or (predicate &rest predicates)` *function*

predicate that is a logical or of predicates

`parallel-iterator (funs time &optional results (max time))` *function*

result of applying transformations in parall

`sequential-iterator (funs time &optional results)` *function*

result of applying transformations sequentically

`second-element (list)` *function*

second element of list

`start-of (timed-note)` *function*

start-time of timed-note data structure

`loudness (note)` *function*

the amplitude of note

`duration (note)` *function*

the duration of note

`first-structure-of (musical-object)` *function*

the first element of a musical object

`max-number-of-parallel-voices (musical-object)` *function*

the maximum number of parallel voices in musical object

`count-one (note)` *function*

the number 1

`number-of-notes (musical-object)` *function*

the number of notes in musical object

`pitch (note)` *function*

the pitch of note

`second-structure-of (complex-structure)` *function*

the second element of a musical object

`clip (min value max)` *function*

the value, clipped within [min,max]

`third-element (list)` *function*

third element of list

`sounding-note? (note)` *function*

true if amplitude of note is non-zero, false otherwise

`note? (object)` *function*

true if note, false otherwise

`middle-c? (note)` *function*

true if pitch of note is 60, false otherwise

`same-note? (note-1 note-2)` *function*

true when notes are equal, false otherwise

`between? (value min max)` *function*

true when value in interval <min, max], false otherwise

`no-note-transform (note)` *function*

untransformed note

`pitch-to-frequency (pitch)` *function*

a MIDI number into a frequency (in Hz.)

Literature references made

(Barendregt, 198?)

(IMA, 1983)

Glossary references made

Object Oriented Programming

abstraction

accumulating parameter

anonymous functions

application

body

combinators

consing

data abstraction

downward funargs

first class citizens

function quote

iteration

lambda-list keywords

lexical scope

list

list iterator

list recursion

numeric recursion

parameter-freeprogramming

part-of hierarchy

predicate

program generators

recursion

side- effect

special form

stack

tail recursion

time function

upward funargs

Text references made

To do

these trees should be eval-hierarchies

these trees should be eval-hierarchies

iets over interactie hier

in (draw (transform ....)) voorbeelden, orginal-and-transform doen?

ongeveer hier iets over een car/cdr recursie? En, symmetrisch, fun op elements and sub-elements, transpose uitschrijven?

vertaal: detwaalfde machtswortel uit twee

[an error occurred while processing this directive]

```
[an error occurred while processing this directive]
```

Lisp as a second language, composing programs and music.

Peter Desain and Henkjan Honing

Chapter III Object-Oriented Style I

Draft mars 11, 1997

1. Defining classes
2. Creating objects
3. Reading slots
4. More classes
5. Naming pitches
6. Methods
7. Generic functions vs. message passing
8. Defining an object representing a rest
9. Transformations
10. Writing slots
11. Transposing whole musical objects
12. Inheritance
13. Initial slot values
14. Classes and types
15. Customizing build-in behavior of the Lisp system
16. Copying objects
17. A metric grammar
18. Reflecting pitches
19. Iterating over part-of structures
20. Searching through part-of structures
21. Slot access vs. methods
22. Setf methods
23. Mapping over the parts of an object
24. Offset
25. Iteration and onset times
26. Draw
27. Time dependent transformations
28. Searching through part-of structures while maintaining onset times
29. Mapping and onset times
30. Before and after methods

31. Definitions made
32. Literature references made
33. Glossary references made

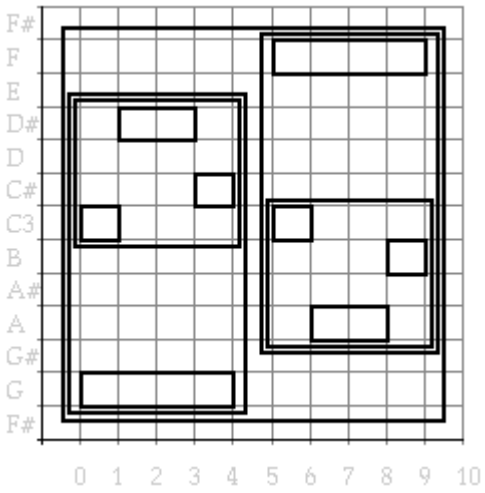

Figure 20. front page

[to do: In intro OOI zowel MIDI keynumbers uitgelegd als primitive keuze “on basis of esthetic or pragmeatic grounds” genoemd]

[to do: use of e.g. in text (forexample, for instance intext, e.g., tussen haakjes) e.g. - e.g., of , e.g.,]

[to do: use of “:” just before code example.]

[to do: :WARNING, : RULE-OF-THUMB, :SOUNDBYTES styles]

[to do: meer voorlezen van code bv defclass form]

[to do: grapje over klasse bewustzijn, engelsen of marx]

Next to Smalltalk, Lisp has been one of the earliest languages to support an *object-oriented style*, and many ideas, that can now be found in object- oriented extensions of traditional languages (like C++) have developed over the years within the Lisp community. The ideas have come to a consistent and mature definition in CLOS, the Common Lisp Object System. This language replaces many of the precursors (Flavors, Common Loops) and has become widely accepted by users of Common Lisp. It contains, next to standard object-oriented mechanisms (e.g., *instantiation* and *inheritance*) more sophisticated concepts (like *multiple inheritance*, *method combination*, *multi-methods* and the *meta object protocol*) that can become powerful tools in constructing computer music systems and representing musical knowledge. The complexity of the full object oriented extension to Common Lisp can be quite overwhelming, especially when one tries to learn it directly from a concise reference such as Steele (1990)

. Therefore this chapter takes a step by step approach with many small examples, which are constructed for the sake of clarity. The exposé of CLOS constructs is divided into three parts. This first part covers all that is needed to define classes, create objects, and define simple methods that can act on them. It contains

enough explanation of the mundane issues to learn to program in a simple object oriented style. The second part will treat more advanced topics like multi-methods and method combination which makes CLOS so much more powerful than other object-oriented languages. After mastering this part the reader will be familiar will all aspects of CLOS that one needs for programming in this language. The third part will show how the CLOS language can be extended in CLOS itself. This meta-approach makes it possible to achieve elegant solutions even when the constructs and mechanisms (e.g., the kind of inheritance) built into the language cannot be used directly for representing the domain in question. Although everyday pedestrian use of CLOS does not often entail working at this level, the concepts of the meta object protocol are of such beauty and power that it will be worthwhile to get acquainted with them a bit. this chapter starts by introducing CLOS and building musical examples from scratch, many Lisp constructs, explained in previous chapters are now assumed to be known. Among them are recursion, the loop macro, lists and arrays, functional arguments, and assignment.

Defining classes

When one uses a general data structure like a *list* to store information, one has to program a *data abstraction* barrier. This set of *selector functions* and *constructor functions* provides access to parts of the data structure, or builds one, and prevents the programmer from having to remember what e.g the `caddar` of a certain list represents. In CLOS the main data structure is the object, a *record*-like concept. The access functions for objects are automatically defined by the system when the type of such an object is defined. There is a rigorous division between an object (an *instance*) and the type of that object (a *class*).[1] In a class definition the programmer declares what attributes (*slots*) an object of that class has and how they are to be accessed. the next example a class `note` is defined for the representation of a simple note of a specific pitch and duration. [2] It has two slots, a duration (in arbitrary units, e.g., seconds or quarter notes) and a key number[3]class definition is made with a `defclass` form. Like all `def...` forms this form has some syntax to remember. First the name of the class to be defined is given, in this case `note`. Following is a list of classes to inherit from (inheritance will be described later), in this case it is empty. After that, a list of *slot descriptors* follows. Each slot descriptor is a list of the slot name and some keyword-value pairs that define the name of the *accessor function*, the name of the *initialization keyword* by which the slot can be given its initial value and the documentation string describing the slot. The last component of the `defclass` form are the *class options*. In this case the only class option is a documentation string which describes the class.

```
(defclass note ()

((duration :accessor duration

:initarg :duration

:documentation "Duration in time units")

(pitch :accessor pitch

:initarg :pitch

:documentation "Pitch in MIDI numbers"))

(:documentation "Class of note objects with pitch and duration attributes"))
```

Creating objects

After having defined the class `note` there is no note object yet: a defclass form is like a type definition. We will have to make an object by instantiating the class using the function `make- instance`. The first argument to `make- instance` is the name of a class and the initialization arguments. Thus we can supply

values for each attribute, each slot of the note, by listing the corresponding *initializationkeyword* supplied in the slot descriptor of the class definition, and a value for that slot.

```
(make-instance 'note :duration 2 :pitch 60)
```

The expression above will return an object that represents a middle-C note of two time units. This object has its two slots initialized accordingly. If the expression above is too elaborate for your taste or, a better reason, if you want to create notes in your program without fixing yourself to this particular class definition and *initializationprotocol* yet, a constructor function has to be defined. Contrary to a `record` definition, a CLOS class definition does not give rise to the automatic generation of a constructur function. Our first attempt at constructing one may look like:

```
(defun note (&key duration pitch)

"Return a note object with a specified duration and pitch"

(make-instance 'note :duration duration :pitch pitch))
```

This `note` constructor function passes its two arguments to `make- instance`. But to flexibly allow for later extensions, we will define a constructor function that passes its whole argument list, whatever that may be, directly to `make- instance`. It uses the `&rest` *lambda list keyword* by which an arbitrary number of arguments in the function call are gathered in a list. `apply` takes care to feed the elements of this list as separate arguments to :Lisp make-instance

.

```
(defun note (&rest attributes)

"Return a note object with specified attributes (keyword value pairs)"

(apply #'make-instance 'note attributes))
```

The fact that `note` is used both as the name of a user defined Lisp function, and of a user defined CLOS class does not cause *name clashes*: it is always clear from the context which construct is intended. Now a note object can be created by calling the `note` function. As shown after the arrow (->) this evaluates to an object of the `note` class printed in a not very readable format.

```
(note :duration 2 :pitch 60)

->

#<NOTE #x3F5BC1>
```

Later we will show how we can program a `play` function that can, e.g., transmit the note over MIDI to a

connected synthesizer and make it audible.

Reading slots

A note object can be accessed (its slots read) by the accessor functions named in the slot descriptors of the class definition of `note`. The next example shows the creation of a note with the new constructur function and some subsequent slot access to print its duration and pitch attributes (the double arrow (=>) means “outputs by side effect” or “prints”).

```
(let ((object (note :duration 3 :pitch 65)))

(print (duration object))

(print (pitch object))

object)

=>

3

65

->

#<NOTE #x3F5FB9>
```

To avoid having to print attributes of notes all the time, for inspection of their values, one can define an external representation of notes in a readable format that directly shows the values of the slots. Here we define a function called `show- note`. It uses the accessor functions to translate the note and the values of its slots into a readable list. We arbitrarily choose to let it yield the same simple list format that happens to be the construction call [4]. **[to do: read macros for note syntax, load forms]**We will need to call this function explicitly for the moment, to show the results of our programming. However, we will tie this function [5] more closely into the system later, such that notes will be printed automatically in this format.

```
(defun show-note (object)

"Return a list describing a note object with its attributes"

(list 'note :duration (duration object) :pitch (pitch object)))


(show-note (note :duration 2 :pitch 61)) -> (note :duration 2 :pitch 61)
```

More classes

We will enrich our domain of musical objects a bit and, as example, add some classes that handle the representation of time order. You might have noticed that a note object has no slot specifying the time of its onset. We will indeed opt for a relative specification of time: the actual onset of a note will be dictated by its context. There are two obvious time relations that two musical objects can have: they can start at the same time or one can start after the other. Specifying the time relations between objects as being either parallel or sequential[6] and building larger musical objects in that way is an alternative to the use of

absolute start times found in most composition systems, but the choice is made here quite arbitrarily - just to illustrate the definition and use of these constructs[7]. Below the definition of two classes is given, one to represent sequential and the other to represent simultaneous musical aggregates. They have only one slot that will be bound to a list of their parts.

```
(defclass sequential ()

((elements :accessor elements

:initarg :elements

:documentation "The successive components"))

(:documentation "A group of musical objects, one after another"))


(defclass parallel ()

((elements :accessor elements

:initarg :elements

:documentation "The simultaneous components"))

(:documentation "A group of musical objects, starting at the same time"))
```

After the class definitions we can define the constructor functions for these compound musical objects.

```
(defun sequential (&rest elements)

"Return a sequential musical object consisting of the specified components"

(make-instance 'sequential :elements elements))


(defun parallel (&rest elements)

"Return a parallel musical object consisting of the specified components"

(make-instance 'parallel :elements elements))
```

Now more elaborate musical objects can be constructed, like a low long note parallel to a sequence of three higher notes [8].

```
(defun example ()

"Returns a simple structured musical object"

(parallel (sequential (note :duration 1 :pitch 60)

(note :duration 2 :pitch 63)
```

```
(note :duration 1 :pitch 61))

(note :duration 4 :pitch 55)))
```

Figure 21. Score of the example.

[to do: waar is deze pict gebleven?]

The score [9] of this example is given in Figure ?. A piano-roll notation of the same musical object is given in Figure ?. On the horizontal axis time runs from left to right, in time units. On the vertical axis pitch is represented. Each rectangle [10]represents a note.

```
(draw (example)) =>
```

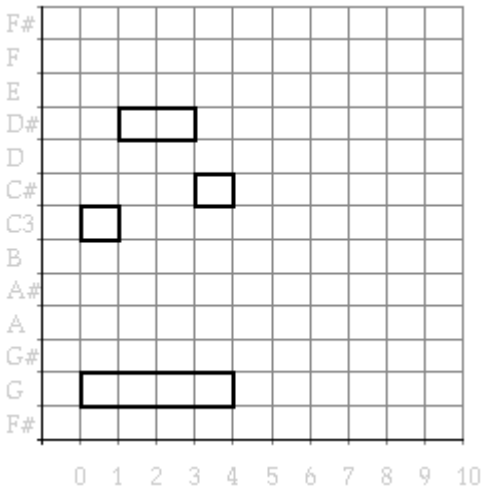

Figure 22. Piano-roll notation of the example.

When the body of `example` is run, three note objects will be created and collected in a list which is stored in the elements slot of an encompassing sequential object. A list of this sequential object together with a fourth lower note is stored in the elements slot of the enclosing parallel object. Thus a structured (parallel or sequential) musical object has access to its parts: they are listed as elements. The musical structures define a *part-of hierarchy* of linked musical objects. This is illustrated in the tree diagram in Figure ?, which constitutes an alternative graphical representation of the same example.

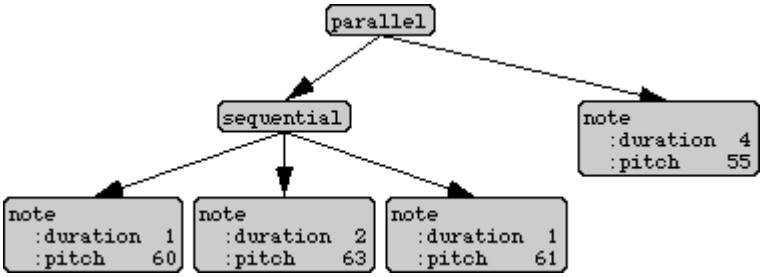

Figure 23. The objects and their links in the example.

In this diagram the arrow represents the "has-part" relation. It reflects the fact that a sequential or parallel object can get access to its parts via its `elements` slot, but not vise versa.

Naming pitches

To make the pitches in our examples a bit easier to read we can translate pitch names to MIDI numbers and vise versa. The two functions `pitch- name- to- midi- number` and `midi- number- to- pitch- name` are given as a separate project in section 5.?: pitch names. **[to do: make this chapter]** They do simple string processing and constitute nothing interesting[11] from the point of CLOS constructs. Let us first check their workings.

```
(pitch-name-to-midi-number "C#4") -> 73

(midi-number-to-pitch-name 60) -> "C3"
```

The `show- note` function can easily be adapted to do the translation and show pitches by name. We can integrate the translation into the `note` function by changing the pitch argument in the *initialization argument list* passed to `make- instance`. This may , e.g., be done destructively, as is shown in the new function definition of `note`. The small effort needed to define a constructor function for notes now already has proven its worth, by allowing for this addition.

[to do: als er een soort non-destructive modify/substitute is, uit vorig hoofdstuk, op property lists dan die gebruilken]

```
(defun note (&rest attributes)

"Return a note object with specified attributes (keyword value pairs)"

(when (stringp (getf attributes :pitch))

(setf (getf attributes :pitch)

(pitch-name-to-midi-number (getf attributes :pitch))))

(apply #'make-instance 'note attributes))
```

```
(defun show-note (object)

"Return a list describing a note object with its attributes"

(list 'note

:duration (duration object)

:pitch (midi-number-to-pitch-name (pitch object))))


(pitch (note :duration 2 :pitch "C#3"))

->

61


(show-note (note :duration 2 :pitch "C#3"))

->

(note :duration 2 :pitch "C#3")
```

One might wonder why we didn't store pitch names directly in the pitch slot of the note. There wouldn't be anything fundamentally wrong with this, except for the fact that the calculations on pitches that we are going to write later (like transpositions) are much easier to implement on a numeric representation.[12]

Methods

Now we will introduce one of the corner stones of object-oriented programming: *polymorphism*. We start with the observation that our structured musical objects have no readable representation yet.

```
(example)

->

#<PARALLEL #xEC7439>
```

To be able to show the new objects in a readable form on the output as well, one could be tempted to define `show- sequential` and `show- parallel` functions, similar to the `show- note` function, to construct the appropriate lists. In calling these functions the programmer then has to be aware of the type of musical objects to show and select the proper function. E.g., when the musical object is parallel (like `example`) `show- parallel` has to be applied. A better approach would be to write a general `show` function that can be applied to any type of musical object, obtains the type (class) of its argument and dispatches accordingly to the appropriate code. This could also enable the function to recursively call because they can have elements of an arbitrary type. This approach is illustrated in the next example.

```
(defun show (object)
```

```
"Return a list describing a musical object with its components or attributes"

(typecase object

(note (list 'note

:duration (duration object)

:pitch (midi-number-to-pitch-name (pitch object))))

(sequential (cons 'sequential

(mapcar #'show (elements object))))

(parallel (cons 'parallel

(mapcar #'show (elements object))))))
```

And indeed, the construction of a so called *polymorphic* or *generic* `show` function that can be applied to many types of musical objects is part of the answer. The first step is to write a CLOS generic function, effectively the same as the function written above, but with some different syntax: now each clause to be executed for a specific class of object is separated out as a so called *method*. For each method the argument list is given again, but annotating the argument with its type. Don't be put off by the syntactical complexity of this form: we will shortly have the system assemble this monster for us. [13]

```
(fmakunbound 'show)


(defgeneric show (object)

(:method ((object note))

(list 'note

:duration (duration object)

:pitch (midi-number-to-pitch-name (pitch object))))

(:method ((object sequential))

(cons 'sequential

(mapcar #'show (elements object))))

(:method ((object parallel))

(cons 'parallel

(mapcar #'show (elements object))))

(:documentation "Return a list describing a musical object"))
```

One of the central ideas of object-oriented programming is the notion of automatic type-driven dispatching, in which the system assembles definitions like the one above from small parts of code written by the programmer. How does this work? We just have to define a set of functions all with the same name, but with different type information for the arguments. These are called methods. Thus three different `show`

methods are to be defined by us, all with the same name but with their argument declared to be of a different class.

```
(defmethod show ((object note))
"Return a list describing a note object with its attributes"
(list 'note
:duration (duration object)
:pitch (midi-number-to-pitch-name (pitch object))))

(defmethod show ((object sequential))
"Return a list describing a sequential musical object with its components"
(cons 'sequential
(mapcar #'show (elements object))))

(defmethod show ((object parallel))
"Return a list describing a parallel musical object with its components"
(cons 'parallel
(mapcar #'show (elements object))))
```

These three chuncks of code achieve the same effect as the bulky `degeneric` form. Silently the system has, when these method definitions are evaluated, assembled a so called generic function `show` similar to the definition of `show- object` given above, but calling the appropriate user-defined `show` methods. The three `show` methods are said to be specialized to the `note`, `sequential`, and `parallel` class respectively. The `show` method for a note returns a list with the attributes, with the atom `note` in front. The `show` method for a sequential or parallel object collects the result of showing its elements in a list and places the proper atom in front. We should test the `show` method now, which is done more easily by pretty-printing its result.

```
(pprint (show (example)))
=>
(parallel (sequential (note :duration 1 :pitch "C3")
(note :duration 2 :pitch "D#3")
(note :duration 1 :pitch "C#3"))
(note :duration 4 :pitch "G2"))
```

Generic functions vs. message passing

Note how the `show` methods for sequential and parallel objects call themselves recursively using a functional argument to `mapcar` to calculate the result for each element. Because in CLOS the individual methods are combined into a generic function automatically, the well-known functional programming techniques like recursion, and passing functions as arguments to other functions, can be applied directly for use on methods. This unification of functional and object oriented programming using generic functions is a large advantage over the *message passing* style. In the latter paradigm objects are being send messages (a special language construct) by other objects. This means that all facilities and constructs developed for functions (mapping, combinators, tracing, etc.) have to be redefined to be useful for messages. In CLOS, methods are applied to objects just like functions are applied to arguments, and all the functional paraphernalia works as well for methods.

Defining an object representing a rest

In a call to `show`, the so called *applicable method* is selected by the system, on the basis of the class of the argument. This has the added advantage that when a program grows and more classes are added, the programmer does not have to modify the functions to handle these data types as well, only additional methods have to be defined. We will demonstrate this by adding a class for a rest and the constructor for it (which will be called `pause` to prevent a *name clash* with the Lisp `rest` function which gives the `cdr` of lists).

```
(defclass pause ()

((duration :accessor duration

:initarg :duration

:documentation "Duration in time units"))

(:documentation "Class of rest objects with a duration attribute"))


(defun pause (&rest attributes)

"Return a rest object with specified attributes (keyword/value pairs)"

(apply #'make-instance 'pause attributes))
```

A rest only needs a duration attribute[14]. Adding the following simple method definition will enable compound musical objects that contain rests to be shown as well:

```
(defmethod show ((object pause))

"Return a list describing a rest object with its attributes"

(list 'pause :duration (duration object)))
```

Transformations

Having set up a simple system of musical objects, a way to create them, and a way to print them textually, we can proceed in defining some useful musical transformations on these objects. The next example defines a method that takes a note as argument and creates a new one with a pitch that is transposed by an

interval. This method is specialized in one argument, the object to transpose. The second argument of the transpose method, the interval, is not typed (i.e. any type of object will apply). In the example a C# is transposed by a whole tone.

```
(defmethod transpose ((object note) interval)

"Return a note with a pitch raised by interval"

(note :duration (duration object)

:pitch (+ (pitch object) interval)))
```

```
(show (transpose (note :duration 2 :pitch "C#") 2))

-> (note :duration 2 :pitch "D#")
```

The example above did instantiate a new `note` object with its duration slot copied directly from the original. This is a proper (non destructive) functional style of programming. But when the original is no longer needed somewhere else, it may make sense to alter one slot in the original itself, instead of creating a new object. And indeed the *imperative style* of programming in which objects are modified destructively is often used in object oriented programming. In this style one has to take much more care that an object that is altered is not accessible from parts of the program that assume that its state is invariant.

Writing slots

The slots of an object can be destructively written by the generalized `setf` *assignment* construct using the same accessors as are used for reading. The next example transposes a note by altering its pitch slot.

```
(defmethod transpose ((object note) interval)

"Raise the pitch of a note by an interval"

(setf (pitch object) (+ (pitch object) interval)))
```

Of course the predefined *modification macro's* that are based on `setf`, like `incf`, work as well for slots.

```
(defmethod transpose ((object note) interval)

"Raise the pitch of a note by an interval"

(incf (pitch object) interval))
```

To illustrate the workings of the transpose method we show a note before and after transposition.

```
(let ((object (note :duration 2 :pitch "C#3")))
```

```
(print (show object))

(transpose object 3)

(print (show object))

t)

=>

(note :duration 2 :pitch "C#3")

(note :duration 2 :pitch "E3")
```

Because `transpose` works by side-effect, it is not really necessary to make it return a value. But often a nicer style of code can result when these destructive transformations do return their modified argument as result.[15] This is a specific style of writing destructively: a function may destroy its arguments but cannot be trusted to update them meaningfully. However, it returns a proper value. This style resembles the protocol that many Common Lisp functions use (e.g., `sort`). It stands in contrast to the style that uses procedures to modify arguments directly, and whose return value can be ignored.

```
(defmethod transpose ((object note) interval)

"Raise the pitch of a note by an interval"

(incf (pitch object) interval)

object)


(show (transpose (note :duration 2 :pitch "C#") 3))

->

(note :duration 2 :pitch "E3")
```

Transposing whole musical objects

Now, of course, we can define the other methods for `transpose` as well, using recursion to transpose the components of sequential and parallel musical objects.

```
(defmethod transpose ((object pause) interval)

"Transpose a rest, do nothing to it"

object)


(defmethod transpose ((object sequential) interval)

"Transpose a sequential musical structure, by transposing its components"

(loop for element in (elements object)

do (transpose element interval))
```

```
object)


(defmethod transpose ((object parallel) interval)
"Transpose a parallel musical structure, by transposing its components"
(loop for element in (elements object)
do (transpose element interval))
object)
```

The compiler will warn that in the `transpose` method for `pause` the interval argument is not used. A *declaration* can instruct the compiler[16], to ignore this parameter and will suppress the warning.

```
(defmethod transpose ((object pause) interval)
"Transpose a rest, do nothing to it"
(declare (ignore interval))
object)
```

With our generic definition of `transpose` and the `show` facility we can now check its working.

```
(pprint (show (transpose (example) 2)))
=>
(parallel (sequential (note :duration 1 :pitch "D3")
(note :duration 2 :pitch "F3")
(note :duration 1 :pitch "D#3"))
(note :duration 4 :pitch "A2"))
```

This transformation can of course also be checked by playing the results using the `play` function. In Figure ? the original example plus its transposition is shown, collected together in a sequential order. Figure ? gives the corresponding piano roll notation.

Figure 24. Score of the example and a transposition over a whole tone.

```
(draw (sequential

(example)

(pause :duration 1)

(transpose (example) 2))) =>
```

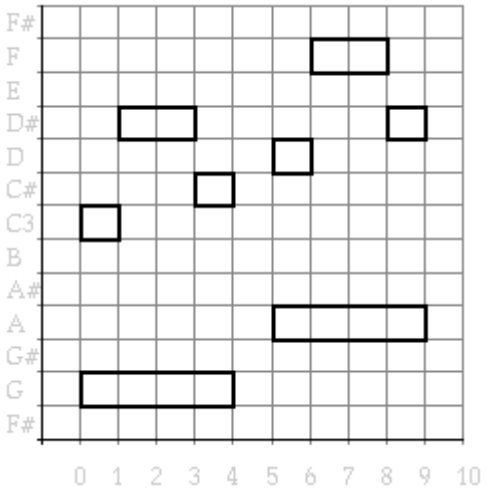

Figure 25. The example and its transposition over a whole tone.

Inheritance

You may have wondered whether `sequential` and `parallel` really need their own definitions of `transpose`, even though the body of the method is exactly the same. The same holds for the duplicated definition of the `elements` slot in the `sequential` and `parallel` class. An indeed, in the object-oriented style one strives towards sharing duplicated code as much as possible. This is done is by making the sharable part explicit as a new *abstract class* that we will call `compound- musical- object`. It has its own slot definition and a `transpose` method defined for it. An abstract class is not intended for direct initialization; it only groups the shared behavior (in this case a slot definition and the `transpose` method) together, such that other classes can inherit this behavior without defining it themselves.

```
(defclass compound-musical-object ()

((elements :accessor elements

:initarg :elements

:documentation "The components"))

(:documentation "A group of musical objects ordered in time"))


(defmethod transpose ((object compound-musical-object) interval)

"Transpose a compound musical structure, by transposing its components"
```

```
(loop for element in (elements object)

do (transpose element interval))

object)
```

Inheriting this class' behavior is done by supplying the class in the list of *direct superclasses* in the `defclass` form of `sequential` and `parallel`. This is the first list after the method's name, the list that unto now had been left empty.

```
(defclass sequential (compound-musical-object) ()

(:documentation "A group of musical objects, one after another"))


(defclass parallel (compound-musical-object) ()

(:documentation "A group of musical objects, starting at the same time"))
```

These definitions show that the duplicated slot descriptors in the definition of the `sequential` and `parallel` class are indeed no longer necessary. The duplicated `transpose` methods of these two classes have become redundant as well. A graphical representation of the inheritance in the form of a class hierarchy is quite useful, especially when the situation becomes more complex. For the moment we have just 2 concrete classes which inherit from the same abstract one. In Figure ? these classes are shown with the arrows representing their directly-inherits-from relation. This is the inverse of the is-direct-superclass-of relation. The concrete classes (that can be instantiated directly) are shown in a gray shade.

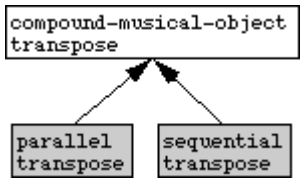

Figure 26. Inheriting transposition behavior for compound musical objects.

The inheritance diagram above is annotated with the `transpose` method defined for these different classes. We see that, next to the new general `transpose` method that was defined for all compound musical objects, the now redundant old `transpose` definitions for parallel and sequential object are still around. In the selection of an applicable method the so called *most specific method* will be chosen to run. This is, roughly spoken, the one encountered first when one starts at the class of the object in the inheritance hierarchy and moves upward. Thus any parallel or sequential object will still be transposed with its old definition that was defined for that specific class, not with the method specialized for the less specific compound musical objects. The old methods are said to *shadow* the new one. In general the use of shadowing can be correct and intended by the programmer-, but often it signals a flaw in the design of the

classes -an inelegant partitioning of knowledge, and it is hard to keep track of the complexities of inheritance when shadowing is used a lot. In our case the unintended shadowing resulted because old method definitions were still around. Removing these old method definitions in de editor textually does not remove them from the system. The difference between the textual state in the editor and the internal state of the system, which can be a source of great confusion, is managed best [17] by keeping all method definitions together and starting afresh after these kinds of large changes. This is done by removing all old method definitions using `fmakunbound`, before evaluating new ones.

```
(fmakunbound 'transpose)


(defmethod transpose ((object compound-musical-object) interval)
"Transpose a compound musical structure, by transposing its components"
(loop for element in (elements object)
do (transpose element interval))
object)


(defmethod transpose ((object pause) interval)
"Transpose a rest, do nothing to it"
(declare (ignore interval))
object)


(defmethod transpose ((object note) interval)
"Raise the pitch of a note by interval"
(incf (pitch object) interval)
object)
```

Thus the definition of the `transpose` method for sequential or parallel objects is disposed of, because it is handled by the `transpose` method specialized for `compound- musical- object`, and the system will find this method to apply to an object of type sequential or parallel because no other, more specific, methods are available. We can similarly cleanup the definition of `note` and `pause` to inherit their shared `duration` slot from a new class for basic (non-compound) musical objects. And it even makes sense to provide an abstract class `musical- object` for (yet unforeseen) behavior that will be shared by all musical objects. This leads to the following class definitions. [18]

```
(defclass musical-object () ()
(:documentation "The abstract root class of all musical objects"))


(defclass compound-musical-object (musical-object)
```

```
((elements :accessor elements

:initarg :elements

:documentation "The components"))

(:documentation "A group of musical objects ordered in time"))


(defclass sequential (compound-musical-object) ()

(:documentation "A group of musical objects, one after another"))


(defclass parallel (compound-musical-object) ()

(:documentation "A group of musical objects, starting at the same time"))


(defclass basic-musical-object (musical-object)

((duration :accessor duration

:initarg :duration

:initform 1

:documentation "Duration in time units"))

(:documentation "A primitive musical object with a duration attribute"))


(defclass note (basic-musical-object)

((pitch :accessor pitch

:initarg :pitch

:initform 60

:documentation "Pitch in MIDI numbers"))

(:documentation "A pitched note"))


(defclass pause (basic-musical-object) ()

(:documentation "A rest"))
```

Now we have created a more complex *class hierarchy* which reflects our view of musical objects and their shared characteristics. It can be depicted as a directed a-cyclic IS-A network of directly-inherits-from relations, as in Figure ?. In this hierarchy, each class inherits only from one other class. More elaborate examples of inheritance will be given later. Note that this figure depicts the general IS-A relation between classes, in contrast to Figure ? which shows the HAS-PART pointers between real instantiated objects.

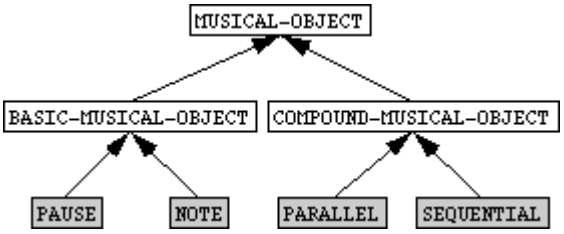

Figure 27. The hierarchy of classes for musical objects.

Again we can show some method definitions in this tree to see where the knowledge [19] needed to show and transpose different types of musical objects resides. Figure ? illustrates how each instantiable class has its own `show` method. `pause` and `note` have their own `transpose` definition, but whenever `transpose` is applied to a sequential or parallel object, the applicable method selected by the system to run, is the method definition specialized to compound musical objects.

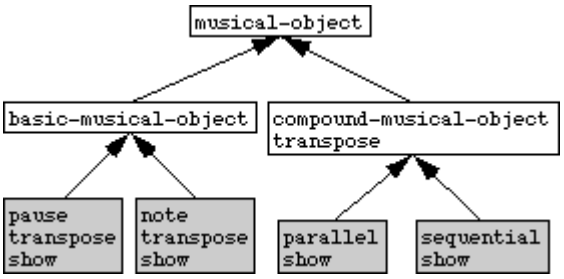

Figure 28. The method definitions of transpose and show.

Initial slot values

Now we will reveal a bit more of the protocol that governs how a new object is instantiated and initialized. In the slot descriptors given in the class definitions above, you may have noticed a new slot option `:initform`. This indicates that when an instance is made (with `make- instance`) without supplying an initialization argument for a certain slot, this value will be used instead. It gives a default value for the slot, e.g., the `duration` slots default to 1 second. Because our `note` and `pause` constructor functions pass their whole argument list to `make- instance`, they do not interfere with this mechanism. And any incomplete call to create a musical object will result in an object with all of its slots bound, some by specification, some by default.

```
(show (pause)) -> (pause :duration 1)

(show (note :duration 4)) -> (note :duration 4 :pitch "C3")
```

This is the first place where we encounter the possibility of declaring default values. In CLOS there are many mechanisms like these available, and the understanding of their proper use, and of their interaction can be quite difficult. We save this topic for the next chapter.

Classes and types

Lisp looks like an *untyped language*: the programmer never has to declare the type of a variable as in other languages. But actually all Lisp data structures carry their type with them and all primitive functions check the type of their arguments before they are run. Thus e.g., the application of `1+` to a list will neatly trigger an error. It is thus better to say that Lisp is a *dynamically typed language*. In constrast to most language which are *statically typed* and that check for type correctnness at *compile time*, the type checks are performed at *run time*. The penalty of run time type checking (slower execution) has been used often as an argument against the use of Lisp. For Common Lisp this argument has become completely unappropriate, since by annotating functions with type declarations the compiler will do the type checking at compile time, and leave out the run time checks. Thus the programmer can quickly write untyped code and, after debugging, add the type declaration which will make it run faster. **[to do: This topic is elaborated further in ??]**

Before CLOS existed, Common Lisp had already an elaborated type system for data structures like numbers, lists, records and arrays. The new class construct of CLOS had to be integrated somehow into this system, this was done by having any CLOS class definition automatically imply the definition of a new Lisp type. Objects that are instantiations of a certain class thus automatically have a certain Lisp type and can e.g. be tested using the `type- of` and `typep` functions.

```
(type-of (example))

->

parallel
```

```
(typep (example) 'note)

->

nil
```

This allows for another definition of the predicates to test a musical object for its class. Let us list the type predicates for all of our concrete (instantiable) classes.Later we will introduce a way to have the class definition define these methods automatically, much like record type definitions do. **[to do: verwijzing naar meta-object-auto-type-prediactes in de voetnoot zetten]**

Because it might be usefull to check any lisp object if it is of a certain class, we specialize on type `t`, the type of any lisp object, numbers, lists and instantiations alike. Instead of specializing the argument to t, we could have left it unspecialized as well or we could have defined a function instead of a method, these forms are similar. As a final refinement we can have the predicate return something useful instead of t. Because in Lisp anything non-`nil` is considered true, this refinement is *backwards compatible* with old definitions which returned `t` or `nil`, while extending the usfulness, e.g., in search functions.

```
(defmethod note? ((object t))
"Return the object if it is an instantiation of class note, nil otherwise"
(when (typep object 'note) object))

(defmethod pause? ((object t))
"Return the object if it is an instantiation of class pause, nil otherwise"
(when (typep object 'pause) object))

(defmethod sequential? ((object t))
"Return the object if it is an instantiation of class sequential, nil otherwise"
(when (typep object 'sequential) object))

(defmethod pause? ((object t))
"Return the object if it is an instantiation of class parallel, nil otherwise"
(when (typep object 'parallel) object))
```

One last remark, before we continue defining more interesting methods: whenever your code makes calls to these type predicates, you might not be using the object oriented style to its full extend. Since the system can automatically execute different pieces of code depending on the type of objects, there is often no need to write addditional code that calls type predicates in deciding which code to run.

Customizing build-in behavior of the Lisp system

It would be nice if our external representation in list form, as produced by the `show` method, could be used for all printing (instead of the unreadable #<...> format). The Common Lisp system already incorporates a `print- object` method for the predefined data types (array's, records etc.) which is called in all printing. We can link our own way of showing the musical objects directly into the system by defining a `print-object` method for `musical- object`, which takes care of pretty-printing as well.

```
(defmethod print-object ((object musical-object) stream)
"Prints a description of a musical object as a list to a stream"
(pprint (show object) stream))
```

After this, all printing is done via our `print- object` method, be it by calls to `print` or `format`, in traces and error messages, or in the top-level *read-eval-print* loop in which the user types an expression to the interpreter and the evaluated result is printed by the system:[20]

```
(example)
```

```
->

(parallel (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

(note :duration 4 :pitch "G2"))
```

This illustrates an important advantage of object-oriented style: the ease of extending code of which the source is not available - we even elaborated a system-defined primitive here. It also shows a constraint on this approach: the argument list of the definition of a method must be known, because they must be similar for all methods. That is, all method definitions with the same name have to have *congruent lambda lists* in which only the types of the argument (and the names) may vary. Further treatment of this topic is given in Steele (1990, page 791)

.

The intuition, that a class which can handle things common to all kinds of musical objects might come in handy one time, thus proved to be right: the `print- object` method is specialized to this general class. The skill to design class hierarchies for a domain, such that later extensions fit in easily, can only be developed during the construction and re-construction of many object oriented programs. Or, in other words, while object oriented programming languages are very flexible and dynamic, and code can often be re-used and extended instead of being re-written, the modification of class hierarchies can entail many complex changes throughout the program. And thus a proper design of these definitions from the start, with enough hooks for adding unforeseen behavior later, is an essential habit.

Copying objects

Because the transformations on musical objects are done destructively, we need a facility to copy a musical object, in case the original is needed as well. To realize this, we simply extract the attributes of a note and create a new note with the same attributes, and make copies of the elements for sequential and parallel objects.[21]

```
(defmethod copy-object ((object note))

"Return a copy of a note object"

(note :duration (duration object) :pitch (pitch object)))


(defmethod copy-object ((object pause))

"Return a copy of a pause object"

(pause :duration (duration object)))


(defmethod copy-object ((object sequential))

"Return a copy of a sequential object, recursively copying its elements"
```

```
(apply #'sequential (mapcar #'copy-object (elements object))))


(defmethod copy-object ((object parallel))

"Return a copy of a parallel object, recursively copying its elements"

(apply #'parallel (mapcar #'copy-object (elements object))))
```

Alternatively, since we already defined a `show` method, which delivers a description that resembles the expression that created the musical object directly, copying can be implemented as evaluating the result of the `show` method:[22]

```
(fmakunbound 'copy-object)


(defmethod copy-object ((object musical-object))

(eval (show object)))
```

We will now continue with defining some useful transformations, using the programs we have made so far. We will write a function that applies an arbitrary transformation to a musical object, and returns the result preceded by the original and separator (e.g., a rest):

```
(defmethod original-and-transform ((object musical-object)

separator transform &rest args)

"Return a sequence of the original, a separator, and a transformed object"

(let ((result (apply transform (copy-object object) args)))

(if separator

(sequential object separator result)

(sequential object result))))


(original-and-transform (note :duration 1 :pitch "C3")

(pause :duration 1)

#'transpose 2)

->

(sequential (note :duration 1 :pitch "C3")

(pause :duration 1)

(note :duration 1 :pitch "D3"))
```

Other functions that combine musical objects can be written as well, e.g., functions to create a repetition and a canon. The results of these functions can be safely subjected to destructive transformations because duplicated objects are copied: there is no shared structure.

```
(defmethod repeat ((object musical-object) &optional (n 2) separator)
"Return a number of repetions of a musical object"
(apply #'sequential
object
(loop repeat (1- n)
when separator collect (copy-object separator)
collect (copy-object object))))

(defmethod canon ((object musical-object) n onset-delay)
"Return an n-voiced canon of a musical object"
(apply #'parallel
(loop for voice from 0 below n
collect (sequential
(pause :duration (* voice onset-delay))
(copy-object object)))))
```

These functions can now be used to create a familiar canon. [23]

[to do: tekst en noten checken]

```
(defun frere-jacques ()
(sequential (note :pitch "C3")
(note :pitch "D3")
(note :pitch "E3")
(note :pitch "C3")))

(defun dormez-vous ()
(sequential (note :pitch "E3")
(note :pitch "F3")
```

```
(note :pitch "G3" :duration 2)))


(defun sonner-la-matine ()
(sequential (note :pitch "G3" :duration 1/2)
(note :pitch "A3" :duration 1/2)
(note :pitch "G3" :duration 1/2)
(note :pitch "F3" :duration 1/2)
(note :pitch "E3")
(note :pitch "C3")))


(defun bim-bam-bom ()
(sequential (note :pitch "C2")
(note :pitch "G2")
(note :pitch "C2" :duration 2)))


(defun frere-jacques-voice ()
(sequential (repeat (frere-jacques))
(repeat (dormez-vous))
(repeat (sonner-la-matine))
(repeat (bim-bam-bom))))


(defun frere-jacques-canon ()
(canon (frere-jacques-voice) 4 8))
```

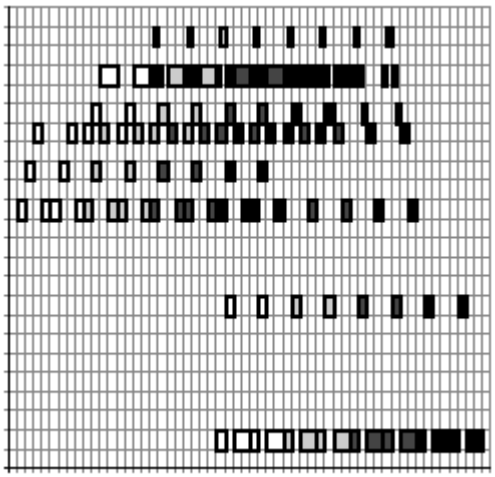

Figure 29. The Frere Jacques canon.

A metric grammar

[to do: could go somewhere else to enlighthen, when needed: gebruikt geen OO of afhankelijkheden]

To give some hints how generative aspects can be implemented elegantly within our framework, we will write some functions that produce a rhythm that is strictly metric. That is, every note starts of one of the levels of the metric hierarchy. Or, to say it in other words, every level is either completely divided or presented as one whole.[24] We first write a program that recursively divides a time duration, using a different divisor at each level, and deciding at random whether subdivision has to continue. This function can be said to embody some kind of generative grammar, deciding at random which of the alternative generative rules (i.e., set of divisors) to use. A `toss` function, that returns `t` for a specified fraction of the times that it is called, is a handy building block.

```
(defun toss (chance)

"There is a chance that this will return t"

(< (random 10000) (* chance 10000)))
```

```
(defun strictly-metric-division (duration divisors)

"return a list of strictly metric durations"

(if (and divisors (toss .75))

(loop repeat (first divisors)

append (strictly-metric-division (/ duration (first divisors))

(rest divisors)))

(list duration)))
```

Figure ? shows the time interval division as produced by the `strictly- metric- division` function.

```
(strictly-metric-division 12 '(2 3 2))

->

(1 1 2 1 1 6)
```

Figure 30. A typical metric division of time intervals

[to do: beter snappen waarom tree draw niet precies goed de sizes uitrekent, is nu een truc]

The results of the `strictly- metric- division` have to be mapped to a sequence of notes. Generating a bar of notes in a certain time-signature can now simply be described as a bar of a certain duration and its subdivision (at each level) given as arguments to the strictly metric generator.

```
(defun make-rhythm (durations)

"Return a sequence of notes with given durations"

(apply #'sequential (loop for duration in durations

collect (note :duration duration))))


(defun bar-strict-3/4 ()

"Return a bar of notes in strict 3/4"

(make-rhythm (strictly-metric-division 3/4 '(3 2 2))))


(defun bar-strict-6/8 ()

"Return a bar of notes in strict 6/8"

(make-rhythm (strictly-metric-division 6/8 '(2 3 2))))


(bar-strict-6/8)

->

(sequential (note :duration 3/8 :pitch "C3")

(note :duration 1/16 :pitch "C3")

(note :duration 1/16 :pitch "C3")

(note :duration 1/8 :pitch "C3")

(note :duration 1/16 :pitch "C3")

(note :duration 1/16 :pitch "C3"))
```

[to do: Hier een keer een score generen met 6/8 indicatie en beaming en vast weergeven (:eval? nil) met de aanroep (play (bar-strict-6/8))]

Reflecting pitches

To study another transformation, and see how it differs from `transposition`, consider a transformation that inverts a piece: reflecting the pitches of all the notes around a pitch center. For a compound musical objects this amounts to recursively applying the mirror transformation to all its elements, for a note the method needs to calculate the reflected pitch and update the pitch slot of the note, for a rest the method will do nothing.

```
(defmethod mirror ((object compound-musical-object) center)
"Reflect all pitches occurring in a compound musical object around a center"
(loop for element in (elements object)
do (mirror element center))
object)

(defmethod mirror ((object note) center)
"Reflect the pitch of a note around a center"
(setf (pitch object)
(- center
(- (pitch object) center)))
object)

(defmethod mirror ((object pause) center)
"Reflect a rest: do nothing to it"
(declare (ignore center))
object)
```

In Figure ? the result of this transformation is shown, applied to the example and preceded by the original.

```
(draw (original-and-transform
(example)
(pause :duration 1)
#'mirror 60)) =>
```

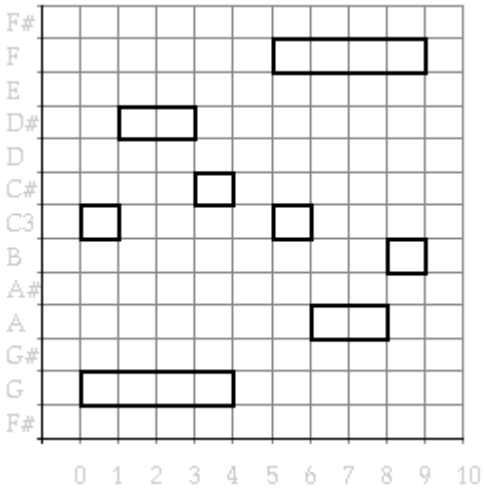

Figure 31. The example and an inversion around middle-c.

The body of the `mirror` method for notes still is a bit too complex to our taste.[25] An auxiliary function can easily do the arithmetic on pitches. A natural name for this function would be `mirror` of course. But then we need to make it into a real method and specialize it for a certain kind of data. Because CLOS is well-integrated in the Lisp type system, we can use ordinary types as specializers for methods as well. This means that on any Lisp data type (e.g., number, string or array) specialized methods can be defined, just as easy as for real classes. We will use this facility to write a `mirror` method specialized for numbers, and call that method inside the method specialized for notes.

```
(defmethod mirror ((pitch number) center)

"Reflect a pitch around a center"

(- center (- pitch center)))


(defmethod mirror ((object note) center)

"Reflect the pitch of a note around a center"

(setf (pitch object) (mirror (pitch object) center))

object)
```

To allow naming of center pitches we need to check for strings and translate them into numbers before we apply the arithmetic.[26]

```
(defmethod mirror ((pitch number) center-pitch)
```

```
"Reflect a pitch around a center"

(let ((center (if (stringp center-pitch)

(pitch-name-to-midi-number center-pitch)

center-pitch)))

(- center (- pitch center))))


(defmethod mirror ((object note) center)

"Reflect the pitch of a note around a center"

(setf (pitch object)

(mirror (pitch object) center))

object)
```

There is one last sideline that we will make before we continuing with CLOS. We noticed that in the definition of `transpose`, `incf` was used to update the pitch slot of an object with the sum of its old value and a specified interval. And instead of `(setf (pitch object)(+ (pitch object) interval)` we wrote `(incf (pitch object) interval)`. Of course this mechanism is what we need here as well: a `mirrorf` macro to update a *generalizedvariable*, like a slot, directly with our user-defined `mirror` method for numbers. Writing the macro to achieve this is not trivial, even though it may look otherwise.[27] However, Common Lisp provides a facility that can define the macro for us. The `define- modify-macro` macro needs a name, a list of extra arguments, a function that calculates a new value, some extra arguments, and a documentation string.

```
(define-modify-macro mirrorf (center)

mirror

"Change a variable by reflecting it around a certain center")
```

Now we may use this modification macro in the body of the `mirror` method for notes.[28]

```
(defmethod mirror ((object note) center)

"Reflect the pitch of a note around a center"

(mirrorf (pitch object) center)

object)
```

Figure 32. The example and its inversion.

Iterating over part-of structures

When working with structures of objects that are linked to one another (like via the `elements` slot of the compound musical objects), one tends to find that many programs exhibit more or less the same way of visiting the different objects of such a structure. Compare, e.g., the `mirror` program and the definition of `transpose`, repeated below.

```
(defmethod mirror ((object compound-musical-object) center)
"Reflect all pitches occurring in a compound musical object around a center"
(loop for element in (elements object)
do (mirror element center))
object)
```

```
(defmethod transpose ((object compound-musical-object) interval)
"Transpose a compound musical structure, by transposing its components"
(loop for element in (elements object)
do (transpose element interval))
object)
```

You can see that both have the same control structure. To extract that functionality we could write something that is like `mapc` is for lists: a general mapping function that applies the operation to each element of a compound musical object. To add some more generality, we will have the new function pass any extra arguments that it receives to the operation, as well.

```
(defmethod mapc-elements ((object compound-musical-object) operation &rest args)
"Apply the operation to each element in a musical object"
(mapc #'(lambda(element)(apply operation element args))
(elements object))
object)
```

`mapc- elements`, when applied to a compound musical object, applies an operation to each of its elements. Of course it could have been written as well using the `loop` construct, the choice is a matter of taste. We will choose to use the later construct in these situations.

```
(defmethod mapc-elements ((object compound-musical-object) operation &rest args)
"Apply the operation to each element in a musical object"
```

```
(loop for element in (elements object)

do (apply operation element args))

object)
```

Thus we can define `transpose` and `mirror` (and a whole bunch of similar transformations) much more concise.

```
(defmethod transpose ((object compound-musical-object) interval)

"Transpose all pitches occurring in a musical object over an interval"

(mapc-elements object #'transpose interval))


(defmethod mirror ((object compound-musical-object) center)

"Mirror all pitches occurring in a musical object around a center"

(mapc-elements object #'mirror center))
```

Searching through part-of structures

A different kind of iterator is needed when the walk through the elements may be terminated prematurely, and a result has to be returned. This typically is the case in functions that search through a structure and return `T` as soon as an encountered object satisfies some predicate. As an example we will test whether our `example` musical object contains a note as one of its elements (it does).

```
(defmethod find-element ((object compound-musical-object) predicate &rest args)

"Return the first element which satifies a predicate, NIL if none do"

(loop for element in (elements object)

thereis (apply predicate element args)))


(find-element (example)

#'note?)

->

(note :duration 4 :pitch "G2")
```

Of course the predicates can be more complex and may even be constructed on the fly.

```
(find-element (example)

(compose-and #'note?
```

```
#'(lambda(note)

(> (duration note) 3))))

->

T
```

In the definition above we cleverly returned the value returned by the predicate, whenever it is not `NIL`. This allows for a predicate that returns some meaningfull value whenever it is satisfied,, e.g., the object that satisfies it. In the next example such a predicate is composed by adding the `identity` function as the last component.

```
(find-element (example)

(compose-and #'note?

#'(lambda(object)(> (duration object) 3))

#'identity))

->

(note :duration 4 :pitch "G2")
```

Using `find- element` recursively (i.e., not only apply it to the top-level elements of the example) yields a more general search function that can search through as whole musical object and return the first object in the part-of hierarchy that satisfies a certain condition.

```
(defmethod find-musical-object ((object compound-musical-object) predicate

&rest args)

"Return object when it satifies the predicate, or first such sub-..-sub-object"

(or (apply predicate object args)

(apply #'find-element object #'find-musical-object predicate args)))


(defmethod find-musical-object ((object basic-musical-object) predicate

&rest args)

"Return object when it satisfies the predicate"

(apply predicate object args))
```

To test our search function we compose a predicate which will return its argument whenever it is applied to a note object with a pitch above middle C.

```
(find-musical-object (example)

(compose-and #'note?

#'(lambda(note)(> (pitch note) 60))

#'identity))

->

(note :duration 2 :pitch "D#3")
```

Slot access vs. methods

In designing inheritance networks it is always good to check if behavior defined for a specific class can be generalized to others. An example is the `duration` slot of basic musical objects that can be accessed by the `duration` method generated by the system. This accessor was generated automatically because `duration` was defined as an accessor in the slot descriptor in the definition of the `basic- musical-object` class. Because the `duration` accessor is itself a method we can extend its applicability by defining methods to calculate the duration of other musical objects. Calling `duration` on a note or a pause will invoke the proper method and return the value of the slot and calling it on a sequential (or parallel) object should now calculate the sum (or maximum) of the duration of its elements.

```
(defmethod duration ((object sequential))

"Return the duration of a sequential musical object"

(loop for element in (elements object)

sum (duration element)))


(defmethod duration ((object parallel))

"Return the duration of a parallel musical object"

(loop for element in (elements object)

maximize (duration element)))


(duration (example)) -> 4
```

This illustrates the advantages of the uniform way of slot access and method calls. The `duration` method can be applied to any musical object now, without having to remember whether a slot access or a calculation has to be done to retrieve it. This enables easy changes of implementation of the duration concept in the program as well (from stored to calculated, from *procedural* to *declarative*), without having to rewrite the parts that use it.

The same mechanism can be used to get an objects' attribute value in a different format. E.g. the frequency of a note (in Hz.) might be of interest. Writing a method to retrieve it looks the same as a slot accessor from the outside. Thus we need not worry whether pitches or frequencies are stored in a note

slot, they both can be retrieved in the same way.[29] The auxiliary `pitch- to- frequency` arithmetic is given in the full code listing that appears at the end of this chapter.

```
(defmethod frequency ((object note))

"Return the frequency of a note (in Hz.)"

(pitch-to-frequency (pitch object)))


(frequency (note :pitch "C#3")) -> 277.18
```

Setf methods

Next to being able to retrieve the freqency of a note, it may be necessary to set it to a specified value as well. It is not so difficult to write a procedure that establishes that, called say `set- frequency` . But now the programmer has to remember that the frequency of a note is changed with a call to the `set-frequency` procedure, while their pitch has to be updated with an assignment (`setf`) on the slot accessor. It is clear that this situation is not very elegant. The more so because it may have been the result of a low level implementation decision and, worse, it may be one that has to change in future modifications of the underlying data representation. The uniform way of slot access and method calls in reading slots provides a solution that can be generalized to the writing of slots as well. In the definition of the `note` class an accessor was given for the pitch slot. This made the system generate a `pitch` method to read the slot and a method to set the pitch of that slot. The last method is named `(setf pitch)`. This is an extension of the common notion of function names and method names that only allow symbols as names. Upon encountering a `(setf (pitch X) Y)` the method `(setf pitch)` is applied to the arguments `Y` and `X`. The system supplied the `(setf pitch)` method that is specialized for notes, because it was asked to do so in the slot descriptor of pitch. The principle of a programmable standard environment[30] applies here as well. And we can define a so called *setf method*[31] ourselves, specialized for notes. It will be called if in the code a `(setf (pitch X) ..)` expression, is evaluated where `X` is a note. This looks just like a slot update.

```
(defmethod (setf frequency) (value (object note))

"Set the frequency of a note"

(setf (pitch object) (frequency-to-pitch value))

value)
```

Note the confusing fact that the order in the argument list of the `setf` method is first the value and then the object: the reverse from the order in which they appear in the call to `setf`. It is important to let a user-defined `setf` method return its value, such that we may rely on it to behave the same as ordinary assignments. The auxiliary function `frequency- to- pitch` is given in the full code listing at the end of this chapter.

[to do: of in een project ergens?]

Setf methods thus give us the mechanism to solve that problem that we can ask for both the pitch and the frequency of a note but only set the former.

```
(let ((object (note :pitch "C3")))

(setf (frequency object) 440)

object)

->

(note :duration 1 :pitch "A3")
```

A rather beautiful consequence of these simple definition of the `frequency` and `(setf frequency)` method is the fact that all modification macro's work (like **incf** and `mirrorf`) consistently when applied to the frequency of a note, translating the pitch of a note to a frequency, modifying that number in the frequency domain (which is an essentially different operation) and translating the result back to proper pitches. Thus miroring the pitch A1 around the pitch A3 (2 octaves up) give the pitch A5, while miroring the frequency of A1 around the frequency of A3 (a difference of 330 Hz) yields a G4.

```
(let ((object (note :pitch "A1")))

(format t "~%~S ; ~A Hz" (show object) (round (frequency object)))

(mirrorf (pitch object) "A3") ; 440 Hz

(format t "~%~S ; ~A Hz" (show object) (round (frequency object))))

=>

(note :duration 1 :pitch "A1") ; 110 Hz

(note :duration 1 :pitch "A5") ; 1760 Hz


(let ((object (note :pitch "A1")))

(format t "~%~S ; ~A Hz" (show object) (round (frequency object)))

(mirrorf (frequency object) 440) ; A3

(format t "~%~S ; ~A HZ" (show object) (round (frequency object))))

=>

(note :duration 1 :pitch "A1") ; 110 Hz

(note :duration 1 :pitch "G4") ; 770 Hz
```

Of course we now want to be able to create a new note of a specified frequency, this can be done in the

note constructor.

```
(defun note (&rest attributes)
"Return a note object with specified attributes (keyword/value pairs)"
(when (getf attributes :frequency)
(setf (getf attributes :pitch)
(frequency-to-pitch (getf attributes :frequency))))
(when (stringp (getf attributes :pitch))
(setf (getf attributes :pitch)
(pitch-name-to-midi-number (getf attributes :pitch))))
(apply #'make-instance 'note attributes))
```

We will show later how we can modify CLOS' initialization mechanism, to be able to directly supply a new initialization keyword to `make- instance`.

[to do: create a note from scratch with a specification of its frequency using an initialize instance. simpler maken?, of uitstellen tot intialize-instance :after behandeld wordt?]

Mapping over the parts of an object

Some calculations on compound musical objects cannot be programmed with the `mapc- elements` iterator, even though the way in which they visit the parts of a musical object is identical. This is, e.g., the case when side-effects are not appropriate and return values have to be constructed. We will write a simple `map- elements` which is analogous to the `mapc- elements` function (it applies an operation to each element of a compound musical object) but this time it collects the results and uses an extra argument that specifies how they are to be combined. In contrast with `mapcar`, which always combines the results in the list to be returned, this mapper gives some extra flexibility by allowing to specify the method of combination. Supplying list would result in behavior that is like `mapcar` applied to the elements of a musical object, but other methods of combination (like adding) can be used as well.

```
(defmethod map-elements ((object compound-musical-object)
combination operation &rest args)
"Combine the results of the operation applied to each element"
(apply combination
(mapcar #'(lambda (element) (apply operation element args))
(elements object))))
```

We will use this mapper to program the `duration` method anew.

```
(defmethod duration ((object sequential))
"Return the duration of a sequential musical object"
(map-elements object #'+ #'duration))

(defmethod duration ((object parallel))
"Return the duration of a parallel musical object"
(map-elements object #'max #'duration))
```

Which concisely reads that the duration of a sequential structure can be obtained by adding the duration of its elements, and that the duration of a parallel structure is the maximum of the duration of its elements. To further illustrate the mapper's use, we will write a program that counts the number of notes in a musical object.

```
(defmethod note-count ((object compound-musical-object))
"Return the number of notes in this musical object"
(map-elements object #'+ #'note-count))

(defmethod note-count ((object note))
"Return 1, there is one note in this musical object"
1)

(defmethod note-count ((object pause))
"Return 0, there are no notes in this musical object"
0)

(note-count (example)) -> 4
```

A slight variation can count the number of musical objects, including compound ones.

```
(defmethod object-count ((object compound-musical-object))
"Return the number of objects in this musical object"
(1+ (map-elements object
```

```
#'+
#'object-count)))

(defmethod object-count ((object basic-musical-object))
"Return one: the multitude of this one objects"
1)

(object-count (example)) -> 6
```

Another variation counts the number of parallel voices that happen inside a musical object, a function that may come of use when we handle voice allocation for synthesizers with a limited number of oscillators that can generate sound simultaneously.[32]

```
(defmethod voice-count ((object sequential))
"Return the number of parallel voices in this musical object"
(map-elements object #'max #'voice-count))

(defmethod voice-count ((object parallel))
"Return the number of parallel voices in this musical object"
(map-elements object #'+ #'voice-count))

(defmethod voice-count ((object note))
"Return 1, there is one voice needed for this musical object"
1)

(defmethod voice-count ((object pause))
"Return 0, there are no voices needed for this musical object"
0)

(voice-count (example)) -> 2
```

Offset

Because the `duration` method is defined for any musical object, it is a simple exercise to define an `offset` method that calculates when a musical object ends, given its time of onset.

```
(defmethod offset ((object musical-object) onset)

"Return the end time of a musical object"

(+ onset (duration object)))
```

Although the appearance of simple expressions which calculate the offset anywhere in the code would not look that incomprehensible, supplying an interface to the data representation of musical objects which is quite rich, and contains, e.g., functions like `offset`, almost always pays off in unforeseen ways when the data representation has to be extended later. As an example of the use of `offset`, we define some methods to check whether a musical object is sounding at a specific time, or during a given time interval, given its onset.

```
(defmethod sounding-at? ((object musical-object) onset time)

"Is the object sounding at time, when started at onset?"

(<= onset time (offset object onset)))


(defmethod sounding-during? ((object musical-object) onset from to)

"Is the object sounding during time-interval [from, to], when started at onset?"

(interval-overlaps? onset (offset object onset) from to))


(defun interval-overlaps? (begin1 end1 begin2 end2)

"Do the intervals [begin,1 end1] and [begin2, end2] overlap?"

(and (<= begin2 end1)

(>= end2 begin1)))


(sounding-during? (note :duration 2) 0 1 3)

->

t
```

Iteration and onset times

Sometimes transformations need the start time of a note, which is not explicitly stored in the object itself. It is not difficult to define an auxiliary function that calculates the onset times of elements of a compound object, given the onset time of the object itself. Calling this method on a sequential object and its onset time should yield the subsequent onset time of its components, and for a parallel structure will return the list of identical onset times of its components.

```
(defmethod onsets ((object sequential) &optional (onset 0))

"Return the onsets of the components of object, given its own onset"
```

```
(integrate (mapcar #'duration (elements object)) onset))



(defun integrate (durations time)
"Return the running sum of the durations, starting from time"
(when durations
(cons time
(integrate (rest durations) (+ time (first durations))))))

(defmethod onsets ((object parallel) &optional (onset 0))
"Return the onsets of the components of object, given its own onset"
(make-list (length (elements object)) :initial-element onset))
```

[to do: aansluiting met integrate uit functional maken]

To test the methods we can apply them to some simple musical objects.

```
(onsets (sequential (note :duration 1)
(pause :duration 2)
(note :duration 4))
3)
->
(3 4 6)

(onsets (parallel (note :duration 1)
(pause :duration 2)
(note :duration 4))
3)
->
(3 3 3)
```

Extending the iterator slightly, it can keep track of these onset times and supply them as extra argument to an operation. Just as in `mapc- elements` all extra arguments supplied are passed on to the operation itself. This often lifts the need for the contruction of a specialized closure for use as operation.[33]

```
(defmethod mapc-elements-onsets ((object compound-musical-object)

onset

operation

&rest args)

"Apply the operation to each element in a musical object and its onset time"

(loop for element in (elements object)

for element-onset in (onsets object onset)

do (apply operation element element-onset args))

object)
```

A simple test of this mapper can be undertaken, e.g., by having it print the elements and their onset times of a coumpound object.

```
(mapc-elements-onsets (sequential (note :duration 1)

(pause :duration 2)

(note :duration 4))

0

#'(lambda (element onset)

(format t "~%~A ; at ~A" (show element) onset)))

=>

(note :duration 1 :pitch "C3") ; at 0

(pause :duration 2) ; at 1

(note :duration 4 :pitch "C3") ; at 3
```

Using this `onsets` method for parallel structures as well, allows one definition for the `mapc- elements- onsets` method for all compound objects (though rather inefficiently defined). We choose here to sacrifice efficiency a bit in favor of the simplicity of the program. A version of `mapc- elements- onsets` for parallel objects that is more efficient, can always be added.[34] Thus even optimization becomes an activity of adding efficient code for special cases, there is no need to modify existing programs. As such a program will be separated into a part that is essential, elegant, and easy to maintain, and a part that addresses the optimization issues. For now, we will try to extend the elegant part with a drawing program based on this new mapper.

[to do: check volgende plaatjes]

Draw

As an example of the use of `mapc- elements- onsets`, and to gain an easy way to inspect the results of our transformations, we will write a graphical program to draw a piano-roll notation of a musical structure using this iterator. We will assume that a graphical primitive `draw- rectangle` exists that draws a rectangle, given a list of the left, top, right and bottom sides. The window on which the pianoroll is drawn is created by the `make- piano- roll- window` function This window takes care of the drawing of the backdrop of the piano roll (the grid lines and the labels on the axes). **[to do: hier een referentie naar een piano-roll project maken ?]**

```
(defmethod draw (object &rest args)
"Draw a graphical piano-roll representation of a compound musical object"
(let ((window (apply #'make-piano-roll-window args)))
(draw-musical-object object 0 window)
window))

(defmethod draw-musical-object ((object compound-musical-object) onset window)
"Draw a graphical piano-roll representation of a compound musical object"
(mapc-elements-onsets object onset #'draw-musical-object window))

(defmethod draw-musical-object ((object note) onset window)
"Draw a graphical representation of a note (a rectangle)"
(draw-rectangle window (boundary object onset window)))

(defmethod draw-musical-object ((object pause) onset window)
"Draw nothing"
(declare (ignore onset window)))

(defmethod boundary ((object note) onset window)
"Return a list of the sides of a graphical representation of a note"
(declare (ignore window))
(list onset ; left
(+ (pitch object) .5) ; top
(offset object onset) ; right
(- (pitch object) .5))) ; bottom
```

As you can see, drawing a compound musical object amounts to drawing all its elements at the proper position, drawing a note entails drawing a rectangle. Rests are ignored for the moment. This was in fact

the program used to draw Figure ?, Figure ? and Figure ?, as the reader who is using the software that comes with this book can check. Only the grid and the labels on the axes were added. Isn't it surprising how such a simple program can achieve this quite complex task?

In Figure ? the call hierarchy is given for the drawing program. It illustrates which functions (methods) call which others. The mapper itself is left out. The direction of the arrows indicate the "calls" or "invokes" relation, gray arrows mean that the call is made indirectly, through a functional argument that was passed to e.g. a mapper. **[to do: tree draw zo maken dat ook grotere cycles getekend kunnen worden, dan kan hier de mapper zelf ook in het diagram]**

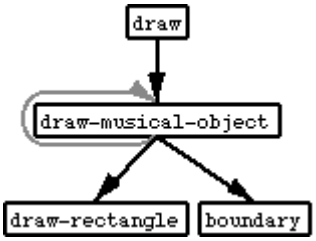

Figure 33. Flow of control in the draw program.

Time dependent transformations

Because `mapc- elements- onsets` neatly administers the onset time of each musical object it has become easy to define time dependent transformations as well. As example let's program a gradual tempo change, using a general time-time map.[35] The `map- time` method is given a musical object, its onset and a function from time to time. It calculates new durations by applying this function to the onset and offset times of all basic musical objects, and updates the duration slots accordingly. The `make- tempo- change` function is given an initial and a final tempo and the duration of the tempo change. It calculates the time-warp function.

```
(defmethod map-time ((object compound-musical-object) onset time-map)

"Apply a time mapping to the notes of the object"

(mapc-elements-onsets object onset #'map-time time-map))


(defmethod map-time ((object basic-musical-object) onset time-map)

"Adjust the duration of a basic musical object according to the time map"
```

```
(let* ((new-onset (funcall time-map onset))

(new-offset (funcall time-map (offset object onset))))

(setf (duration object)(- new-offset new-onset))))
```

[to do: funtion van score naar perf tijd, dan tempo factoren 1/x. laten we er een mooi sundberg ritard van maken]

```
(defun make-tempo-change (duration begin end)

"Return a time-warp function based on begin and end tempo factors"

#'(lambda (time)

(+ (* begin time)

(* (/ (- end begin)

(* 2 duration))

(square time)))))


(defun square (x)

"Return the square of a number"

(* x x))


(draw (map-time (repeat (example) 3 (pause :duration 1))

0

(make-tempo-change 14 1 .33))) =>
```

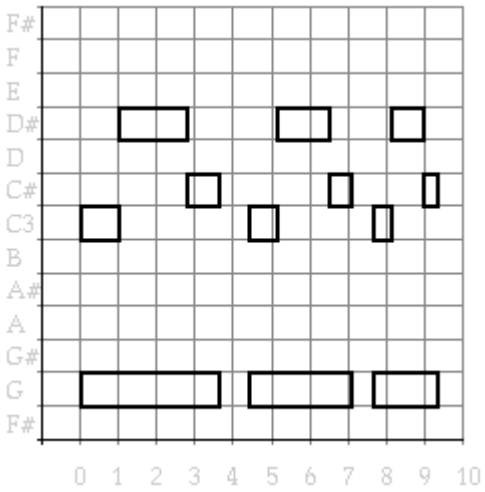

Figure 34. A gradual tempo change.

Searching through part-of structures while maintaining onset times

A refinement of our `find- element` function resembles the way in which `mapc- elements` was extended into `mapc- elements- onsets`. It maintains the time of onset and expects the predicate to have at least one extra argument: the onset of the object that it is testing.

```
(defmethod find-element-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing elements and their onset"

(loop for element in (elements object)

for onset in (onsets object onset)

thereis (apply predicate element onset args)))
```

```
(find-element-onset (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

0

#'(lambda(object onset)(= onset 1)))

->

t
```

The above example searches through the elements of a sequential musical object and returns `t` when it encounters an element starting at time 1. Note the use of `thereis` clause in the body of `find- element- onset`. This does not just return `t` when the predicate yields a non-`nil` value, it returns the value returned by the predicate (just like `and`, when all its arguments are non-`nil`, returns the value of its last argument). This enables us to use not-so boolean predicates that return something meaningful, e.g., the object which satified it. In the example below the first note which starts at 0 is returned. This approach makes extra demands to the set of predicates for musical objects, and the ways in which they can be combined. However, it lifts the need for specialized iterators.

```
(find-element-onset (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

0

#'(lambda(object onset)
```

```
(and (note? object)

(= onset 1)

object)))

->

(note :duration 2 :pitch "D#3")
```

Recursively calling `find- element- onset` yields again a refinement of a more general search function that can search through all levels of a musical object using a subtle mutual recursion of `find- element` and `find- musical- object- onset`, passing each other through the control structure by way of functional arguments.

```
(defmethod find-musical-object-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing sub-objects and their onset"

(or (apply predicate object onset args)

(apply #'find-element-onset object onset

#'find-musical-object-onset predicate args)))


(defmethod find-musical-object-onset ((object basic-musical-object)

onset predicate &rest args)

"Return predicate's result, when satisfied for this object at its onset"

(apply predicate object onset args))
```

To 'read' the code presented above: `find- musical- object- onset`, first tests the predicate and returns its value when non-`nil`. When it is `nil`, and the object under consideration is compound, it calls find-musical-object on itself (passing it aguments as well) to have itself applied to all elements of the object. Note how the use of `&rest` arguments in the definition of both `find- musical- object` and `find- musical- object- onset` allows for the passing of extra parameters that remain invariant. This makes the use of closures to capture the value of those parameters in the functions themselves unneccesary. Note how the predicate itself is shifted in and out of the role of extra argument at the different levels of the recursion: for `find- musical- object- onset` it is a main parameter, but when passed on to `find- element- onset` it becomes an extra one, with `find- musical- object- onset` taking the role of the predicate again. When `find- element- onset` in turn calls `find- musical- object- onset`, it passes its extra parameters, with the predicate in the first place, to `find- musical- object- onset` which thus receives the predicate as a main parameter.

To test our search function we compose a predicate which will return its argument whenever it is applied to a note, its onset, and a time at which the note would be sounding. Because we are composing the `sounding- at?` predicate of three arguments (the object, its onset and a time) with simple ones of only

one argument, we need to wrap those into a function that can be given more arguments, but ignores all abut the first.

[to do: use-only-first-arg-wrap uit functional, of ref naar S K combinators ?]

```
(defun use-first-arg (function)

"Wrap the function such that it can be given spurious arguments"

#'(lambda(&rest args)

(funcall function (first args))))


(find-musical-object-onset (example)

0

(compose-and (use-first-arg #'note?)

#'sounding-at?

(use-first-arg #'identity))

3)

->

(note :duration 2 :pitch "D#3")
```

Please observe that the above search function does not really represent an object oriented style at all: it makes excessive use of functional arguments. It should be kept in mind that the method could have been written in a truely object oriented way with distributed control over the recursion, as we did in the `draw-musical- object` method. However, the solution presented above is quite elegant and we will use this approach a lot. [36]

Mapping and onset times

Of course our `map- elements` should be extended to be able to maintain the onset times of the objects to which the operation is applied as well, just like `mapc- elements` was.

```
(defmethod map-elements-onsets ((object compound-musical-object)

onset operation &rest args)

"Return the results of the operation applied to each element and its onset"

(loop for element in (elements object)

for element-onset in (onsets object onset)

collect (apply operation element element-onset args)))
```

To illustrate the use of this iterator, let us write a method that allows us to collect the set of pitches that is being used in a certain time interval in a musical piece. The function to be used when combining the results of the recursive calls on each element of a compound musical object is `set- union`, which has to be defined by us because Common Lisp only supplies the function to calculate the union of two sets.

```
(defun set-union (&rest args)

"Return the union of any number of sets"

(reduce #'union args))


(defmethod sounding-pitches ((object compound-musical-object) onset from to)

"Return a list of all pitches that occur in time interval [from, to]"

(apply #'set-union (map-elements-onsets object onset #'sounding-pitches from to)))


(defmethod sounding-pitches ((object note) onset from to)

"Return a list of the pitch of the note, when sounding in interval [from, to]"

(when (sounding-during? object onset from to)

(list (pitch object))))


(defmethod collect-sounding-pitches ((object pause) onset from to)

"Return the empty list representing silence"

nil)


(mapcar #'midi-number-to-pitch-name

(sounding-pitches (example) 0

2 4))

->

("C#3" "D#3" "G2")
```

Before and after methods

To illustrate how behavior defined by means of methods can be extended further, let us modify the piano roll drawing by filling each note rectangle with a dark shade. Here again, we would like to add a piece of code instead of rewriting the `draw- musical- object` method. In CLOS, methods come in different qualities (we have only used the *primary methods* unto now). And when more methods are applicable, they are combined by the system using the so called method combination procedures. In our case, the added behavior can operate by *side effect*, and the extra work can be put in a so called *before method* or *after method*. We will assume that a primitive `paint- rectangle` exists to fill a rectangle with a shade and write a before method for `draw- musical- object`. The character of this methods is indicated with the *method qualifier* `:before` that appears between the name and the argument list.

```
(defmethod draw-musical-object :before ((object note) onset window)

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window (boundary object onset window) :light-gray))
```

When a `draw- musical- object` call is to be executed, the system finds all *applicable methods* depending on the class of object that the method is applied to and combines them into a whole, the so called *effective method*. In *standard method combination* all before and after methods and the *primary method* are assembled in their proper order. Any behavior that can be conceptualized as a side effect to take place before or after some main behavior that was already defined, can be added in this way. The use of *declarative method combination* frees the programmer from writing code that searches for methods and calls them in the correct order.

```
(draw (example)) =>
```

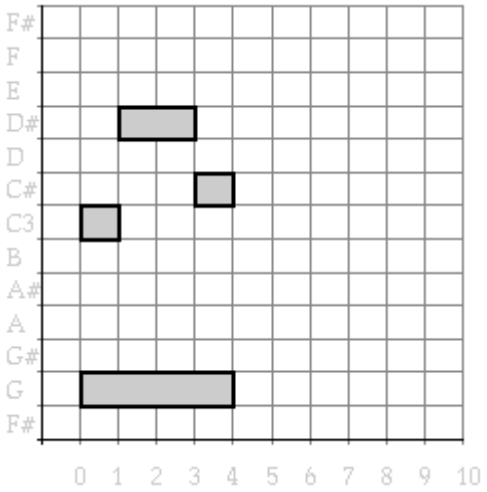

Figure 35. Shaded piano-roll notation of example.

Note that the source code of the main primary method is not changed - it needs not even be available. Thus also predefined behavior, e.g., of other authors ,or of the Lisp system itself, can be modified or elaborated. The tree diagram in Figure ? depicts where the different `draw- musical- object` methods are defined, and which will run in the different situations. `draw- musical- object` has its own definition for a note, a rest and any compound musical object, while one before method exists to do the shading, which is specialized for the class `note`, and will only run before a note is drawn by its primary method.

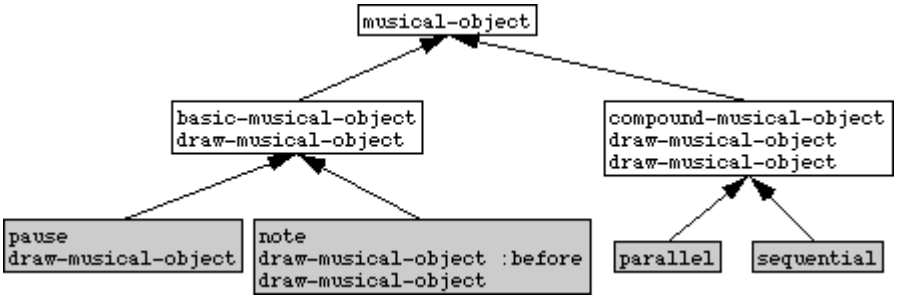

Figure 36. The draw-musical-object method definitions and musical object classes.

[to do: make the methods look different in a different font then the classes (how?)]

Looking for possible optimization of our code, we see that both `draw- musical- object` methods for notes call the `boundary` function, they are repeated below.

```
(defmethod draw-musical-object :before ((object note) onset window)

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window (boundary object onset window) :light-gray))
```

```
(defmethod draw-musical-object ((object note) onset window)

"Draw a graphical representation of a note (a rectangle)"

(draw-rectangle window (boundary object onset window)))
```

Thus, because the before and the primary method cannot communicate information, the `boundary` method is wastefully called twice for each note, once in the before method to draw a shaded rectangle, and once in the primary method to draw the boundaries around it. By defining an auxiliary function (`draw- boundary`), and moving the code that needs the boundary from `draw- musical- object` (the shared code in both functions) to this new function, this duplicated calculation can be prevented. The `draw- boundary` embodies thus the action: draw a note given its boundary box.

```
(defmethod draw-musical-object ((object note) onset window)

"Draw a graphical representation of a note (a rectangle)"

(draw-boundary object (boundary object onset window) window))
```

```
(defmethod draw-boundary ((object note) boundary window)

"Draw a box with a specified boundary (left top right bottom)"
```

```
(draw-rectangle window boundary))


(defmethod draw-boundary :before ((object note) boundary window)
"Fill a graphical representation of a note with a gray color"
(paint-rectangle window boundary :light-gray))
```

Before and after methods form a powerfull way to slice-up behaviour in tiny morsels that can be added, managed and grouped much better than bulky code that has to be rewritten for each modification. The possibility to define more methods specialized to the same class with different qualifiers, and in a declarative way state the order in which they must be called, is one of the features that set CLOS aside from many other object oriented languages.

Definitions made

`compound-musical-object (musical-object)` *`class`*

A group of musical objects ordered in time

`sequential (compound-musical-object)` *`class`*

A group of musical objects, one after another

`parallel (compound-musical-object)` *`class`*

A group of musical objects, starting at the same time

`note (basic-musical-object)` *`class`*

A pitched note

`basic-musical-object (musical-object)` *`class`*

A primitive musical object with a duration attribute

`pause (basic-musical-object)` *`class`*

A rest

`map-time (object onset time-map)` *`method`*

Adjust the duration of a basic musical object according to the time map

`mapc-elements (object operation &rest args)` *`method`*

Apply the operation to each element in a musical object

`mapc-elements-onsets (object onset operation &rest args)` *`method`*

Apply the operation to each element in a musical object and its onset time

`mirrorf (center)` *`macro`*

Change a variable by reflecting it around a certain center

`map-elements (object combination operation &rest args)` *`method`*

Combine the results of the operation applied to each element

`interval-overlaps? (begin1 end1 begin2 end2)` *function*

Do the intervals [begin,1 end1] and [begin2, end2] overlap?

`draw (object &rest args)` *method*

Draw a graphical piano-roll representation of a compound musical object

`draw-musical-object (object onset window)` *method*

Draw a graphical representation of a note (a rectangle)

`draw-boundary (object boundary window)` *method*

Fill a graphical representation of a note with a gray color

`sounding-at? (object onset time)` *method*

Is the object sounding at time, when started at onset?

`sounding-during? (object onset from to)` *method*

Is the object sounding during time-interval [from, to], when started at onset?

`mirror (object center)` *method*

Mirror all pitches occurring in a musical object around a center

`print-object (object stream)` *method*

Prints a description of a musical object as a list to a stream

`note-count (object)` *method*

Return 0, there are no notes in this musical object

`voice-count (object)` *method*

Return 0, there are no voices needed for this musical object

`bar-strict-3/4 nil` *function*

Return a bar of notes in strict 3/4

`bar-strict-6/8 nil` *function*

Return a bar of notes in strict 6/8

`show (object)` *function*

Return a list describing a musical object with its components or attributes

`show-note (object)` *function*

Return a list describing a note object with its attributes

`show (object)` *method*

Return a list describing a rest object with its attributes

`sounding-pitches (object onset from to)` *method*

Return a list of the pitch of the note, when sounding in interval [from, to]

`boundary (object onset window)` *method*

Return a list of the sides of a graphical representation of a note

`note (&rest attributes)` *function*

Return a note object with specified attributes (keyword/value pairs)

`repeat (object &optional n separator)` *method*

Return a number of repetions of a musical object

`parallel (&rest elements)` *function*

Return a parallel musical object consisting of the specified components

`pause (&rest attributes)` *function*

Return a rest object with specified attributes (keyword/value pairs)

`make-rhythm (durations)` *function*

Return a sequence of notes with given durations

`original-and-transform (object separator transform &rest args)` *method*

Return a sequence of the original, a separator, and a transformed object

`sequential (&rest elements)` *function*

Return a sequential musical object consisting of the specified components

`make-tempo-change (duration begin end)` *function*

Return a time-warp function based on begin and end tempo factors

`canon (object n onset-delay)` *method*

Return an n-voiced canon of a musical object

`find-element-onset (object onset predicate &rest args)` *method*

Return first non-nil predicate's result, testing elements and their onset

`find-musical-object (object predicate &rest args)` *method*

Return object when it satisfies the predicate

`object-count (object)` *method*

Return one: the multitude of this one objects

`find-musical-object-onset (object onset predicate &rest args)` *method*

Return predicate's result, when satisfied for this object at its onset

`duration (object)` *method*

Return the duration of a parallel musical object

`collect-sounding-pitches (object onset from to)` *method*

Return the empty list representing silence

`offset (object onset)` *method*

Return the end time of a musical object

`find-element (object predicate &rest args)` *method*

Return the first element which satifies a predicate, NIL if none do

`frequency (object)` *method*

Return the frequency of a note (in Hz.)

`note? (object)` *method*

Return the object if it is an instantiation of class note, nil otherwise

`pause? (object)` *method*

Return the object if it is an instantiation of class parallel, nil otherwise

`sequential? (object)` *method*

Return the object if it is an instantiation of class sequential, nil otherwise

`onsets (object &optional onset)` *method*

Return the onsets of the components of object, given its own onset

`map-elements-onsets (object onset operation &rest args)` *method*

Return the results of the operation applied to each element and its onset

`integrate (durations time)` *function*

Return the running sum of the durations, starting from time

`square (x)` *function*

Return the square of a number

`set-union (&rest args)` *function*

Return the union of any number of sets

`example nil` *function*

Returns a simple structured musical object

`(setf frequency) (value object)` *method*

Set the frequency of a note

`musical-object nil` *class*

The abstract root class of all musical objects

`toss (chance)` *function*

There is a chance that this will return t

`pitch-to-frequency (pitch)` *function*

Translate a MIDI number into a frequency (in Hz.)

`frequency-to-pitch (frequency)` *function*

Translate a frequency (in Hz.) to a MIDI pitch number

`transpose (object interval)` *method*

Transpose all pitches occurring in a musical object over an interval

`use-first-arg (function)` *function*

Wrap the function such that it can be given spurious arguments

`frere-jacques-canon nil` *function*

`frere-jacques-voice nil` *function*

`bim-bam-bom nil` *function*

`sonner-la-matine nil` *function*

`dormez-vous nil` *function*

`frere-jacques nil` *function*

`copy-object (object)` *method*

`strictly-metric-division (duration divisors)` *function*

return a list of strictly metric durations

Literature references made

(Balaban, 1992)

Balaban, M. 1992 Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music. in Balaban, M., K. Ebcioglu & O. Laske (Eds.) Cambridge, Mass.: MIT Press. 111-138

Bamberger (??)

Chowning, 1973 Journal of the Audio Engineering Society, 21(7):526-534. Reprinted inC. Roads and J. Strawn, (Eds.) 1985 Foundations of Computer Music. Cambride, MA: MIT Press. pp. 6-29.

Dannenberg (1993)

Dannenberg, R.B. 1991 Real-time Scheduling and Computer Accompaniment. in Mathews, M.V and J.R. Pierce (Eds.) . Cambridge, MA: MIT Press.

Dannenberg, R.B. 1993 The Implementation of Nyquist, a Sound Synthesis Language. in San Francisco: ICMA.

Dannenberg, R.B., P. Desain and H. Honing (in Press) Programming Language Design for Music. in G. De Poli, A. Picialli, S.T. Pope & C. Roads (Eds.) Lisse: Swets & Zeitlinger.

Desain, P. and H. Honing (In Preparation) CLOSe to the edge, Advanced Object-Oriented Techniques in the Representation of Musical Knowledge. Jounral of New Music Research.

Desain, P. 1990 Lisp as a second language, functional aspects. (28)1 192-222

International MIDI Association 1983 North Holywood: IMA.

Moore, F.R. 1990 Englewood Cliffs, New Jersey: Prentice Hall.

Pope, S.T. 1992 The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers.16(3) 73-91

Pope, S.T. 1993 Machine Tongues XV: Three Packages for Software Sound Synthesis. 17(2) 23-54

Steele (1990)

Steele (1990, page 791)

Steele, G.R. 1990 Bedford MA: Digital Press.

Tatar,D.G. 1987 Bedford, MA: Digital Press.

Glossary references made

abstract class

access function

all functions part of a data abstraction layer (selector and constructor functions). [loaded from Glossary Functional]

accessor function

after method

anonymous function

A function whose ‘pure’ definition is given, not assigning it a name at the same time. [loaded from Glossary Functional]

applicable method

applicable methods

application

obtaining a result by supplying a function with suitable arguments. [loaded from Glossary Functional]

assignment

atom

in Lisp: any symbol, number or other non-list. [loaded from Glossary Functional]

backwards compatible

before method

class

class hierarchy

class options

combinator

A function that has only functions as arguments and returns a function as result. [loaded from Glossary

Functional]

compile time

congruent lambda lists

cons

A Lisp primitive that builds lists. Sometimes used as verb: to add an element to a list. [loaded from Glossary Functional]

constant function

A function that always returns the same value [loaded from Glossary Functional]

constructor function

A function that as part of the data abstraction layer provides a way of building a data structure from its components. [loaded from Glossary Functional]

constructor functions

continuations

A way of specifying what a function should do with its arguments. [loaded from Glossary Functional]

coroutines

parts of the program that run in alternation, but remember their own state of computation in between switches. [loaded from Glossary Functional]

data abstraction

A way of restricting access and hiding detail of data structures [loaded from Glossary Functional]

data abstraction

data type

A class of similar data objects, together with their access functions. [loaded from Glossary Functional]

declaration

declarative

declarative method combination

dialect

A programming language can be split up into dialects that only differ (one hopes) in minor details. Lisp dialects are abundant and may differ a lot from each other even in essential constructs. [loaded from Glossary Functional]

direct superclasses

dynamically typed language

effective method

first class citizens

rule by which any type of object is allowed in any type of programming construct. [loaded from Glossary Functional]

free variables

function

A program or procedure that has no side effects [loaded from Glossary Functional]

function composition

the process of applying one function after another. [loaded from Glossary Functional]

function quote

A construct to capture the correct intended meaning (with respect tothe current lexical environment) of a anonymous function so it can beapplied later in another environment. It is considered good programming style to use function quotes as well when quoting the name of a function. [loaded from Glossary Functional]

functional abstraction (or procedural abstraction)

A way of making a piece of code more general by turning part of it into a parameter, creating a function that can be called with a variety of values for this parameter. [loaded from Glossary Functional]

functional argument (funarg)

A function that is passed as argument to another one (downward funarg) or returned as result from other one (upward funarg) [loaded from Glossary Functional]

generalizedvariable

generic

global variables

an object that can be referred to (inspected, changed) from any part of the program. [loaded from Glossary Functional]

higher order function

A function that has functions as arguments. [loaded from Glossary Functional]

imperative style

inheritance

initializationkeyword

initializationprotocol

initialization argument list

initialization keyword

instance

instantiation

iteration

repeating a certain segment of the program. [loaded from Glossary Functional]

lambda list keyword

lambda-list keyword

A keyword that may occur in the list of parameter names in a function definition. It signals how this function expects its parameters to be passed, if they may be ommited in the call etc. [loaded from Glossary Functional]

lexical scoping

A rule that limits the ‘visibility’ of a variable to a textual chunk of the program. Much confusion can result from the older- so called dynamic scoping - rules. [loaded from Glossary Functional]

list

message passing

meta object protocol

method

method combination

The declaritive way in which CLOS allows more methods to be bundled and run, in situations where more are applicable

method qualifier

methods

modification macro's

most specific method

multi-methods

multiple inheritance

name clash

name clashes

object oriented programming

A style of programming whereby each data type is grouped with its own access function definitions, possibly inheriting them from other types. [loaded from Glossary Functional]

object-oriented style

parameter-free programming

A style of programming whereby only combinators are used to build complex functions from simple ones. [loaded from Glossary Functional]

part-of hierarchy

polymorphic

polymorphism

prefix notation

A way of notating function application by prefixing the arguments with the function. [loaded from Glossary Functional]

primary method

primary methods

procedural

quote

A construct to prevent the Lisp interpreter from evaluating an expression. [loaded from Glossary Functional]

read-eval-print

record

recursion

A method by which a function is allowed to use its own definition. [loaded from Glossary Functional]

run time

selector function

A function that as part of the data abstraction layer provides access to a data structure by returning part of it. [loaded from Glossary Functional]

selector functions

setf method

shadow

side effect

side effect

Any actions of a program that may change the environment and so change the behavior of other programs. [loaded from Glossary Functional]

slot descriptors

slots

stack

A list of function calls that are initiated but have not yet returned a value. [loaded from Glossary Functional]

standard method combination

statically typed

structure preserving

A way of modifying data that keeps the internal construction intact but may change attributes attached to the structure. [loaded from Glossary Functional]

tail recursion

A way of recursion in which the recursive call is the ‘last’ thing the program does. [loaded from Glossary Functional]

untyped language

To do

make the methods look different in a different font then the classes (how?)

use-only-first-arg-wrap uit functional, of ref naar S K combinators ?

funtion van score naar perf tijd, dan tempo factoren 1/x. laten we er een mooi sundberg ritard van maken

tree draw zo maken dat ook grotere cycles getekend kunnen worden, dan kan hierde mapper zelf ook in het diagram

hier een referentie naar een piano-roll project maken ?

check volgende plaatjes

aansluiting met integrate uit functional maken

create a note from scratch with a specification of its frequency using an initialize instance.simpler maken?, of uitstellen tot intialize-instance :after behandeld wordt?

of in een project ergens?

Hier een keer een score generen met 6/8 indicatie en beaming en vast weergeven (:eval? nil) met de aanroep (play (bar-strict-6/8))

beter snappen waarom tree draw niet precies goed de sizes uitrekent, is nu een truc

could go somewhere else to enlighthen, when needed: gebruikt geen OO of afhankelijkheden

tekst en noten checken

verwijzing naar meta-object-auto-type-prediactes in de voetnoot zetten

This topic is elaborated further in ??

als er een soort non-destructive modify/substitute is, uit vorig hoofdstuk, op property lists dan die gebruilken

make this chapter

waar is deze pict gebleven?

read macros for note syntax, load forms

Make a picture of keyboard +pitchnames + midi numbers) add ref in footnote

grapje over klasse bewustzijn, engelsen of marx

meer voorlezen van code bv defclass form

:WARNING, : RULE-OF-THUMB, :SOUNDBYTES styles

use of ":" just before code example.

use of e.g. in text (forexample, for instance intext, e.g., tussen haakjes)e.g. - e.g., of , e.g.,

In intro OOI zowel MIDI keynumbers uitgelegd als primitive keuze "on basis of esthetic or pragmeatic grounds" genoemd

```
[an error occurred while processing this directive]
```

```
[an error occurred while processing this directive]
```

Lisp as a second language,composing programs and music.

Peter Desain and Henkjan Honing

Chapter III Object-Oriented Style II

Draft mars 11, 1997

1. Draw
2. Time dependent transformations
3. Searching through part-of structures while maintaining onset times
4. Mapping and onset times
5. Before and after methods
6. Drawing and the window system
7. Changing structure
8. Maintaining graphical consistency
9. Applying transformations to parts of musical objects
10. Introducing multi-methods
11. A window with a selection
12. Multiple inheritance
13. Binding our own programs to existing menu commands
14. Binding commands to our own menu and control keys
15. multi-methods, method combination and multiple inheritance combined
16. Showing structure
17. Making selections visible, around method
18. Making musical objects mouse sensitive
19. Definitions made
20. Literature references made
21. Glossary references made

Draw

As an example of the use of `mapc- elements- onsets`, and to gain an easy way to inspect the results of our transformations, we will write a graphical program to draw a piano-roll notation of a musical structure using this iterator. We will assume that a graphical primitive `draw- rectangle` exists that draws a rectangle, given a list of the left, top, right and bottom sides. The window on which the pianoroll is drawn is created by the `make- piano- roll- window` function This window takes care of the drawing of the backdrop of the piano roll (the grid lines and the labels on the axes). **[to do: hier een referentie naar een piano-roll project maken ?]**

```
(defmethod draw (object &rest args)
"Draw a graphical piano-roll representation of a compound musical object"
(let ((window (apply #'make-piano-roll-window args)))
(draw-musical-object object 0 window)
window))

(defmethod draw-musical-object ((object compound-musical-object) onset window)
"Draw a graphical piano-roll representation of a compound musical object"
(mapc-elements-onsets object onset #'draw-musical-object window))

(defmethod draw-musical-object ((object note) onset window)
"Draw a graphical representation of a note (a rectangle)"
(draw-rectangle window (boundary object onset window)))

(defmethod draw-musical-object ((object pause) onset window)
"Draw nothing"
(declare (ignore onset window)))

(defmethod boundary ((object note) onset window)
"Return a list of the sides of a graphical representation of a note"
(declare (ignore window))
(list onset ; left
(+ (pitch object) .5) ; top
(offset object onset) ; right
(- (pitch object) .5))) ; bottom
```

As you can see, drawing a compound musical object amounts to drawing all its elements at the proper position, drawing a note entails drawing a rectangle. Rests are ignored for the moment. This was in fact the program used to draw Figure 1, Figure 2 and Figure 3, as the reader who is using the software that comes with this book can check. Only the grid and the labels on the axes were added. Isn't it surprising how such a simple program can achieve this quite complex task?

```
(draw (example)) =>
```

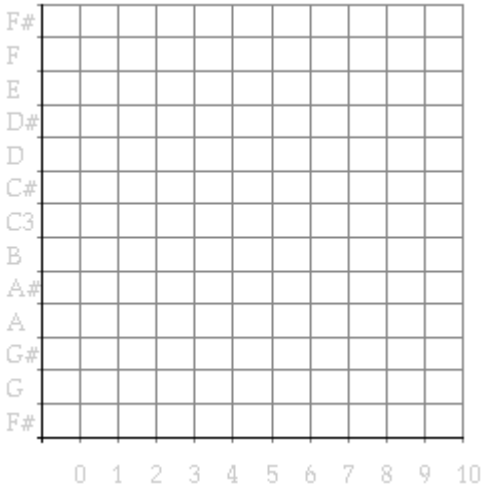

Figure 1. Piano-roll notation of the example.

```
(draw (original-and-transform
(example)
(pause :duration 1)
#'transpose 2)) =>
```

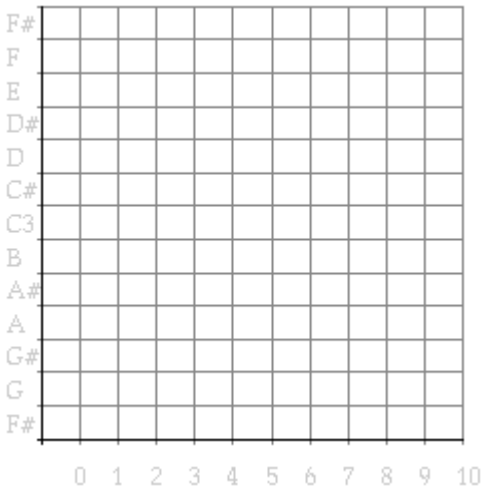

Figure 2. The example and its transposition over a whole tone.

```
(draw (original-and-transform
(example)
```

```
(pause :duration 1)

#'mirror "C3")) =>
```

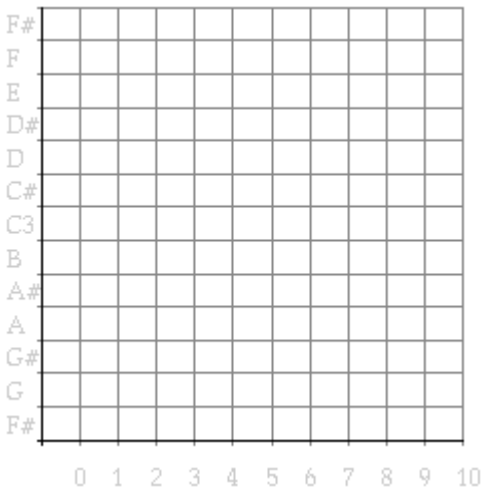

Figure 3. The example and an inversion around middle-c.

In Figure 4 the call hierarchy is given for the drawing program. It illustrates which functions (methods) call which others. The mapper itself is left out. The direction of the arrows indicate the “calls” or “invokes” relation, gray arrows mean that the call is made indirectly, through a functional argument that was passed to e.g. a mapper. **[to do: tree draw zo maken dat ook grotere cycles getekend kunnen worden, dan kan hier de mapper zelf ook in het diagram]**

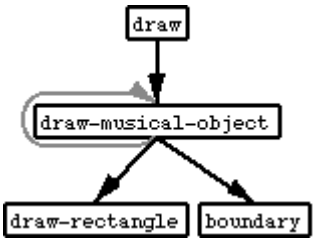

Figure 4. Flow of control in the draw program.

Time dependent transformations

Because `mapc- elements- onsets` neatly administers the onset time of each musical object it has become easy to define time dependent transformations as well. As example let’s program a gradual tempo change, using a general time-time map.[1] The `map- time` method is given a musical object, its onset and a function from time to time. It calculates new durations by applying this function to the onset and offset

times of all basic musical objects, and updates the duration slots accordingly. The `make- tempo- change` function is given an initial and a final tempo and the duration of the tempo change. It calculates the time-warp function.

```
(defmethod map-time ((object compound-musical-object) onset time-map)

"Apply a time mapping to the notes of the object"

(mapc-elements-onsets object onset #'map-time time-map))


(defmethod map-time ((object basic-musical-object) onset time-map)

"Adjust the duration of a basic musical object according to the time map"

(let* ((new-onset (funcall time-map onset))

(new-offset (funcall time-map (offset object onset))))

(setf (duration object)(- new-offset new-onset))))
```

[to do: funtion van score naar perf tijd, dan tempo factoren 1/x. laten we er een mooi sundberg ritard van maken]

```
(defun make-tempo-change (duration begin end)

"Return a time-warp function based on begin and end tempo factors"

#'(lambda (time)

(+ (* begin time)

(* (/ (- end begin)

(* 2 duration))

(square time)))))


(defun square (x)

"Return the square of a number"

(* x x))


(draw (map-time (repeat (example) 3 (pause :duration 1))

0

(make-tempo-change 14 1 .33))) =>
```

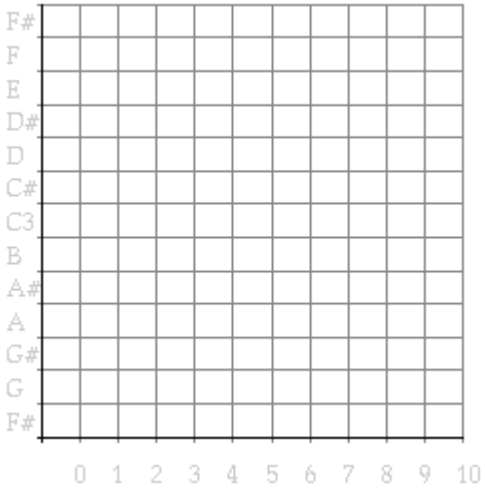

Figure 5. A gradual tempo change.

Searching through part-of structures while maintaining onset times

A refinement of our `find- element` function resembles the way in which `mapc- elements` was extended into `mapc- elements- onsets`. It maintains the time of onset and expects the predicate to have at least one extra argument: the onset of the object that it is testing.

```
(defmethod find-element-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing elements and their onset"

(loop for element in (elements object)

for onset in (onsets object onset)

thereis (apply predicate element onset args)))


(find-element-onset (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

0

#'(lambda(object onset)(= onset 1)))

->

t
```

The above example searches through the elements of a sequential musical object and returns `t` when it encounters an element starting at time 1. Note the use of `thereis` clause in the body of `find-element-onset`. This does not just return `t` when the predicate yields a non-`nil` value, it returns the value returned by the predicate (just like `and`, when all its arguments are non-`nil`, returns the value of its last argument). This enables us to use not-so boolean predicates that return something meaningful, e.g., the object which satified it. In the example below the first note which starts at 0 is returned. This approach makes extra demands to the set of predicates for musical objects, and the ways in which they can be combined. However, it lifts the need for specialized iterators.

```
(find-element-onset (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

0

#'(lambda(object onset)

(and (note? object) (= onset 1) object))

)

->

(note :duration 2 :pitch "D#3")
```

Recursively calling `find-element-onset` yields again a refinement of a more general search function that can search through all levels of a musical object using a subtle mutual recursion of `find-element` and `find-musical-object-onset`, passing each other through the control structure by way of functional arguments.

```
(defmethod find-musical-object-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing sub-objects and their onset"

(or (apply predicate object onset args)

(apply #'find-element-onset object onset

#'find-musical-object-onset predicate args)))


(defmethod find-musical-object-onset ((object basic-musical-object)

onset predicate &rest args)

"Return predicate's result, when satisfied for this object at its onset"

(apply predicate object onset args))
```

To ‘read’ the code presented above: `find-musical-object-onset`, first tests the predicate and returns

its value when non-`nil`. When it is `nil`, and the object under consideration is compound, it calls find-musical-object on itself (passing it aguments as well) to have itself applied to all elements of the object. Note how the use of `&rest` arguments in the definition of both `find- musical- object` and `find- musical- object- onset` allows for the passing of extra parameters that remain invariant. This makes the use of closures to capture the value of those parameters in the functions themselves unneccesary. Note how the predicate itself is shifted in and out of the role of extra argument at the different levels of the recursion: for `find- musical- object- onset` it is a main parameter, but when passed on to `find- element- onset` it becomes an extra one, with `find- musical- object- onset` taking the role of the predicate again. When `find- element- onset` in turn calls `find- musical- object- onset`, it passes its extra parameters, with the predicate in the first place, to `find- musical- object- onset` which thus receives the predicate as a main parameter.

To test our search function we compose a predicate which will return its argument whenever it is applied to a note, its onset, and a time at which the note would be sounding. Because we are composing the `sounding- at?` predicate of three arguments (the object, its onset and a time) with simple ones of only one argument, we need to wrap those into a function that can be given more arguments, but ignores all abut the first.

[to do: use-only-first-arg-wrap uit functional, of ref naar S K combinators ?]

```
(defun use-first-arg (function)

"Wrap the function such that it can be given spurious arguments"

#'(lambda(&rest args)

(funcall function (first args))))


(find-musical-object-onset (example)

0

(compose-and (use-first-arg #'note?)

#'sounding-at?

(use-first-arg #'identity))

3)

->

(note :duration 2 :pitch "D#3")
```

Please observe that the above search function does not really represent an object oriented style at all: it makes excessive use of functional arguments. It should be kept in mind that the method could have been written in a truely object oriented way with distributed control over the recursion, as we did in the `draw- musical- object` method. However, the solution presented above is quite elegant and we will use this approach a lot. [2]

Mapping and onset times

Of course our `map- elements` should be extended to be able to maintain the onset times of the objects to which the operation is applied as well, just like `mapc- elements` was.

```
(defmethod map-elements-onsets ((object compound-musical-object)

onset operation &rest args)

"Return the results of the operation applied to each element and its onset"

(loop for element in (elements object)

for element-onset in (onsets object onset)

collect (apply operation element element-onset args)))
```

To illustrate the use of this iterator, let us write a method that allows us to collect the set of pitches that is being used in a certain time interval in a musical piece. The function to be used when combining the results of the recursive calls on each element of a compound musical object is `set- union`, which has to be defined by us because Common Lisp only supplies the function to calculate the union of two sets.

```
(defun set-union (&rest args)

"Return the union of any number of sets"

(reduce #'union args))


(defmethod sounding-pitches ((object compound-musical-object) onset from to)

"Return a list of all pitches that occur in time interval [from, to]"

(apply #'set-union (map-elements-onsets object onset #'sounding-pitches from to)))


(defmethod sounding-pitches ((object note) onset from to)

"Return a list of the pitch of the note, when sounding in interval [from, to]"

(when (sounding-during? object onset from to)

(list (pitch object))))


(defmethod collect-sounding-pitches ((object pause) onset from to)

"Return the empty list representing silence"

nil)


(mapcar #'midi-number-to-pitch-name

(sounding-pitches (example) 0

2 4))
```

```
->

("C#3" "D#3" "G2")
```

Before and after methods

To illustrate how behavior defined by means of methods can be extended further, let us modify the piano roll drawing by filling each note rectangle with a dark shade. Here again, we would like to add a piece of code instead of rewriting the `draw- musical- object` method. In CLOS, methods come in different qualities (we have only used the *primary methods* unto now). And when more methods are applicable, they are combined by the system using the so called *method combination* procedures. In our case, the added behavior can operate by *side effect*, and the extra work can be put in a so called *before method* or *after method*. We will assume that a primitive `paint- rectangle` exists to fill a rectangle with a shade and write a before method for `draw- musical- object`. The character of this methods is indicated with the *method qualifier* `:before` that appears between the name and the argument list.

```
(defmethod draw-musical-object :before ((object note) onset window)

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window (boundary object onset window) :light-gray))
```

When a `draw- musical- object` call is to be executed, the system finds all *applicable methods* depending on the class of object that the method is applied to and combines them into a whole, the so called *effective method*. In *standard method combination* all before and after methods and the *primary method* are assembled in their proper order. Any behavior that can be conceptualized as a side effect to take place before or after some main behavior that was already defined, can be added in this way. The use of *declarative method combination* frees the programmer from writing code that searches for methods and calls them in the correct order.

```
(draw (example)) =>
```

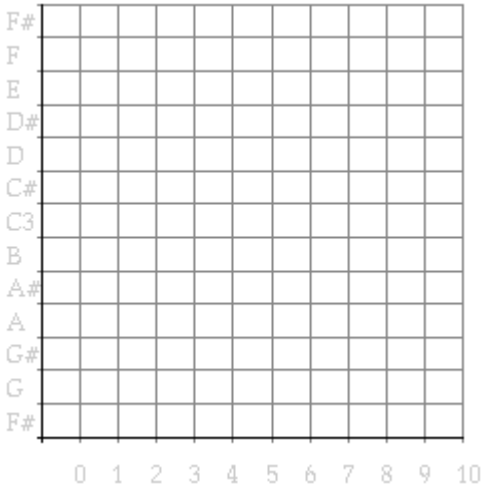

Figure 6. Shaded piano-roll notation of example.

Note that the source code of the main primary method is not changed - it needs not even be available. Thus also predefined behavior, e.g., of other authors ,or of the Lisp system itself, can be modified or elaborated. The tree diagram in Figure 7 depicts where the different `draw- musical- object` methods are defined, and which will run in the different situations. `draw- musical- object` has its own definition for a note, a rest and any compound musical object, while one before method exists to do the shading, which is specialized for the class `note`, and will only run before a note is drawn by its primary method.

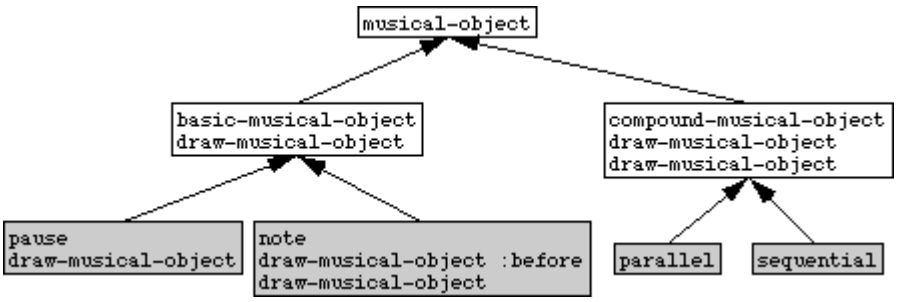

Figure 7. The draw-musical-object method definitions and musical object classes.

[to do: make the methods look different in a different font then the classes (how?)]

Looking for possible optimization of our code, we see that both `draw- musical- object` methods for notes call the `boundary` function, they are repeated below.

```
(defmethod draw-musical-object :before ((object note) onset window)

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window (boundary object onset window) :light-gray))


(defmethod draw-musical-object ((object note) onset window)

"Draw a graphical representation of a note (a rectangle)"

(draw-rectangle window (boundary object onset window)))
```

Thus, because the before and the primary method cannot communicate information, the `boundary` method is wastefully called twice for each note, once in the before method to draw a shaded rectangle, and once in the primary method to draw the boundaries around it. By defining an auxiliary function (`draw- boundary`), and moving the code that needs the boundary from `draw- musical- object` (the shared code in both functions) to this new function, this duplicated calculation can be prevented. The `draw- boundary` embodies thus the action: draw a note given its boundary box.

```
(defmethod draw-musical-object ((object note) onset window)
"Draw a graphical representation of a note (a rectangle)"
(draw-boundary object (boundary object onset window) window))

(defmethod draw-boundary ((object note) boundary window)
"Draw a box with a specified boundary (left top right bottom)"
(draw-rectangle window boundary))

(defmethod draw-boundary :before ((object note) boundary window)
"Fill a graphical representation of a note with a gray color"
(paint-rectangle window boundary :light-gray))
```

Before and after methods form a powerfull way to slice-up behaviour in tiny morsels that can be added, managed and grouped much better than bulky code that has to be rewritten for each modification. The possibility to define more methods specialized to the same class with different qualifiers, and in a declarative way state the order in which they must be called, is one of the features that set CLOS aside from many other object oriented languages.

Drawing and the window system

In the Macintosh window system, drawing directly on a window yield a result that is somewhat volatile because whenever, e.g., in response to a user action, such a window is covered by another one and subsequently uncovered, part of the drawing is gone. In this window system it is the obligation of the window itself (not of the covering window) to maintain a consistent graphical image. In order to allow a window to redraw its contents after such operations the window system calls the `view- draw- contents` method on the window that was just uncovered. We can define a after method for `view- draw- contents` ourselves, to redraw the pianoroll. For that to work the musical object needs to be stored in the window, such that the method (which is called with the window as its sole argument) can get hold of it whenever a redraw is needed. To maintain modularity we define a new window class, based on the piano roll window, that realises this feature. For easy generalization we give it an extra slot with the function that actually does the drawing - such that it will be usable with many drawing programs.

```
(defclass piano-roll-contents-draw-window (piano-roll-window)
((object :accessor object :initarg :object :initform nil)
(draw-function :accessor draw-function :initarg :draw-function :initform nil))
(:documentation "A window for drawing and re-drawing musical objects"))

(defmethod view-draw-contents :after ((window piano-roll-contents-draw-window))
"(Re)draw the stored object as a pianoroll"
```

```
(when (object window)

(funcall (draw-function window) (object window) window)))
```

[to do: uitleg when object conditie]

At window creation time, we now have to supply a musical object, a drawing function. Any drawing and redrawing will be initiated by the window system, as appropriate. Even initial drawing, just after the window is created (like in our original `draw` function), is unnessecary because the window system itself will call `view- draw- contents` on any newly created windows. In this way our drawings are better integrated in the workings of the Macintosh window environment. Again, to allow for future extensions, we add one extra level of inheritance and define a class `draw- window` based on the just defined `piano-roll- contents- draw- window` class, to allow for future window behavior.

```
(defclass draw-window (piano-roll-contents-draw-window)

()

(:documentation "A simple pianoroll window class"))
```

The `draw` method now just has to make an instance of such a window, and store a musical object and a draw function in the appropriate slots (instead of also drawing the object). The invocation of the drawing action itself is taken care of by the window system. Furthermore, to allow the passing of other initialization arguments to the window (such as its title or size), we make the `draw` method accept any extra argument. They are all passed as keyword arguments in a list to the `make- instance` function. In this way, known and future window classes will work with specific initialization arguments not known right now.

```
(defmethod draw (object &rest args &key (window-class 'draw-window) &allow-other-keys)

"Draw a graphical piano-roll representation of a compound musical object"

(apply #'make-instance window-class

:object object

:draw-function #'(lambda(object window)

(draw-musical-object object 0 window))

args))
```

In summary, covering and uncovering (part of) a window will shedule (asynchronously) a `view- draw- contents` command. Whenever that command is executed, first the `view- draw- contents` method of the piano roll window itself is run, which draws the grid. Then our own after method is run, which calls the `draw- musical- object` method on the object stored in the window slot. This finally results in the musical object being redrawn on the screen. To optimize this process, the window system takes care to set the so called view port, the area of the screen that is actually updated in drawing, to only cover the region that has been affected and needs to be redrawn. The drawing primitives will only perform their work in this area.

Changing structure

[to do: deze sectie kan evt naar voren, als voorbeeld van OO-style gedistribueerde recursie]

The main operation in our transformations was applied to notes only, the structure of the musical object remained invariant. This will no longer be the case in the next example: a `retrograde` transformation for musical objects that reverses each sequential (sub-)structure. This transformation destructively modifies only the sequential objects encountered, the others are visited but left untouched. The result of applying this transform is illustrated in Figure 8.

```
(defmethod retrograde ((object parallel))

"Reverse all components of a musical object in time, recursively"

(mapc-elements object #'retrograde)

object)


(defmethod retrograde ((object basic-musical-object))

"Do nothing"

object)


(defmethod retrograde ((object sequential))

"Reverse all components of the object in time"

(setf (elements object)

(reverse (elements object)))

object)


(draw (original-and-transform

(example)

(pause :duration 1)

#'retrograde)) =>
```

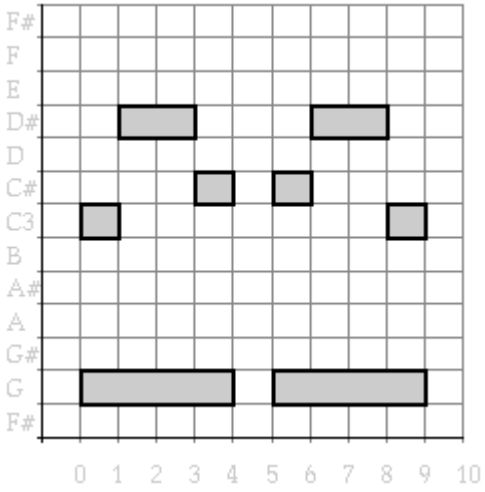

Figure 8. The example and a reversal.

Maintaining graphical consistency

Since musical objects can be stored in windows, we could direct our transformations of musical objects to these windows themselves (accessing them for the moment by title string). The `window- operation` function fist chech whether it is given a window, before it applies the operation to the musical object stored in it, to allow `window- operation` to be called on functions that search a window (like `find-window`) but may return `nil`.

```
(defun window-operation (window operation &rest args)

"Apply the operation to a musical object stored in a window"

(when window

(apply operation (object window) args))

window)
```

[to do: hidden hier vanaf hier het eigen draw window gebruiken voor figuur generatie]

```
(draw (example) :window-title "example musical object")

(window-operation (find-window "example musical object") #'retrograde) =>
```

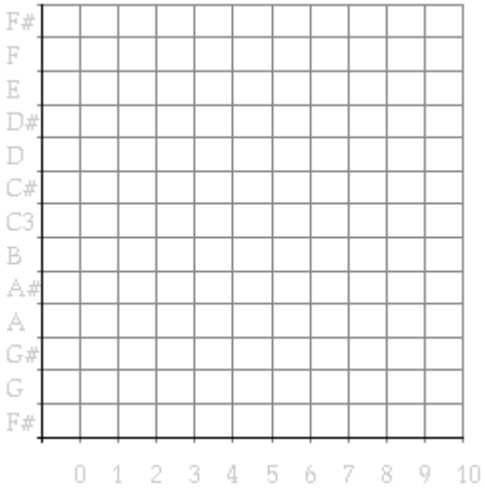

Figure 9. Unconsistency

[to do: `window- operation` method op `piano- roll contents- draw window` ? En slot-name object -> musical-object?]

However, applying a destructive transformation to an object that is displayed on a window yields an inconsistency between the graphical representation and the musical object, e.g. the chance made by the `retrograde` transformation above is not yet shown in the window. Covering the window, by selecting another one or moving another one on top of it, and uncovering it, may force a redraw and reconsitute the consistency, but this is not a very elegant solution. **[to do: recording mixin zo maken dat dit werkt voor de lezer]**It is posible to force a redraw ourselves by calling `invalidate- view`. This method forces the window system to assynchronously (at some time in the future) call `view- draw- contents`, which will redo the drawing. We will use `invalidate- view` to be able to do destructive transforms on objects displayed in a window while maintaining consitency between object and its graphical presentation. The extra argument `t` to `invalidate- view` signals that the window needs to be wiped before any redrawing can be done.

```
(defun window-operation (window operation &rest args)

"Apply the operation to a musical object stored in a window"

(when window

(apply operation (object window) args)

(invalidate-view window t)

window))
```

As long as we call all transforms on musical objects in windows by way of `window- operation` consistency is guaranteed. Instead of using `find- window` we will now assume that the window in which the object of interest is displayed is the front-most of the drawing windows. We can write a `front- draw-window` function that gets hold of that window, bringing it to the front as well (if it wasn't already). Based

on that function it is not difficult to write the `front- window- operation` function which invokes an operation on an object stored in the front most `draw- window`.

```
(defun front-draw-window ()
"Return the frontmost drawing window, bringing it to the front as well"
(let ((window (front-window :class 'draw-window)))
(when window
(window-select window)
window)))

(defun do-front-window (action &rest args)
; onhandig te onthouden, of meer gebruiken hieronder
(let ((window (front-draw-window)))
(when window
(apply action window args)
window)))

(defun front-window-operation (operation &rest args)
"Apply an operation to a musical object stored in the frontmost draw window"
(apply #'window-operation (front-draw-window)
operation
args))
```

Now we can initiate transformations on objects visible in windows as we like it, by bringing the window to the front by a mouse click or using the windows menu. Opperations are issued on the front draw window and the system automatically updates the graphgical representation.

```
(draw (example) :window-titel "example window 1")
(draw (example) :window-titel "example window 2")
; bring one of the window to the front
(front-window-operation #'mirror "C3")
; or the other one
(front-window-operation #'transpose 3)
; and maybe the first again
```

```
(front-window-operation #'retrograde)
```

The operation may, of course, not only do transformations, other actions like, e.g., playing the musical object visible in a window are useful as well.

```
;bring one of the windows with a musical object to the front

(front-window-operation #'play)

(front-window-operation #'play :tempo 240)

(front-window-operation #'retrograde)

(front-window-operation #'play)
```

Applying transformations to parts of musical objects

We have only applied our transformations to complete musical objects, globally, from the outside so to say. However, there is no reason why they shouldn't be used on objects nested (deeply) inside other objects. In Figure 10 we select a part of the example, and apply a transpose operation to it. Our `find-musical- object- onset` function, which searches through an object, comes in handy here.

```
(draw (example)) => 12.A. draw the example

(front-window-operation

#'(lambda(object)

(transpose

(find-musical-object-onset object

0

#'(lambda(part onset)

(and (= onset 1) part)))

3))) => 12.B. Transpose the part which starts at time 1
```

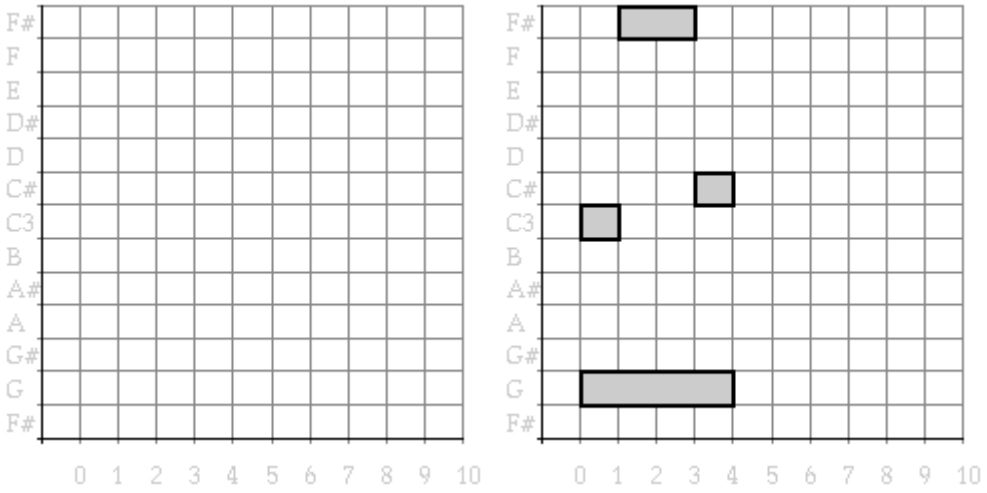

Figure 12. Transposition of the part of the example which starts at time 1

Introducing multi-methods

The method definitions we have written so far all have one argument specialized to a specific class, and in deciding which method is (or which methods are) applicable in a certain case, the method definition specialized to the most specific class of this argument is chosen. This situation, in which a method dispatches on the type of one argument, is still compatible with older object oriented paradigms develloped in LISP and with the *message passing style* of e.g. Smalltalk. In this style, objects handle so called messages (the name of the method plus the other arguments) sent to it, with the message handling selected according to its type. But in some cases, the forced assymetry of deciding which object to sent the message to is quite unnatural. Should we, e.g., write a draw method specialized to different kinds of windows that explicitly inspect the type of the item to be drawn and call the proper procedure, or should we write draw methods specialized to different kinds of grapical items which inspect the type of window to draw on, and call the proper procedures? Or consider an even more symmetrical example, the design of a equality test on musical objects: should this method be specialized to the class of the first object, in the body do a check on the class of the second one, and call the proper procedures? Luckily the answer to these question is: neither. It was a beautiful insight of the designers of CLOS that message passing as view of object oriented programming is not general enough, and that it much better to think of methods as identically named functions with differently typed argument lists, of which a fitting one is selected for each call. Thus in the cases mentioned as examples here it is much better to define methods based on the classes of both their arguments.This dispatching on more arguments, by so called *multi-methods*, boosts the expressivity of CLOS enormously. As a first example, the predicate that tests for equality of musical objects is given below. For each two same-typed objects there is a method defined which checks for equality of elements or attributes. One general method for any two musical objects will catch all other cases, when the predicate is applied to objects that are not of the same class, it can return `nil` immediately.

[to do: Remark: `same?` can't be implemented with our recursor map-elements-onset, since its need two musical objects as argumetn (instead of one). We therefore use athe distributed form of recursion here instead.]

```
(defmethod same? ((object1 sequential)(object2 sequential))
```

```
"Are two sequential musical objects equal?"

(same-list (elements object1)(elements object2) :test #'same?))


(defmethod same? ((object1 parallel)(object2 parallel))

"Are two two parallel musical objects equal?"

(same-set (elements object1)(elements object2) :test #'same?))


(defmethod same? ((object1 note)(object2 note))

"Are two notes equal?"

(and (= (pitch object1) (pitch object2))

(= (duration object1) (duration object2))))


(defmethod same? ((object1 pause)(object2 pause))

"Are two rests are equal?"

(= (duration object1) (duration object2)))


(defmethod same? ((object1 musical-object)(object2 musical-object))

"Return nil as default case: two musical objects of different type are unequal"

nil)
```

For two sequential musical objects to be equal, objects from both lists of elements have to be pairwise equal. For two parallel musical objects equality must hold for the elements of each object, but with the list of elements considered as unordered set. For notes and rests all attributes have to match. For all other combinations `nil` should be returned. Because CLOS always selects *the most specific method* to run, in case where more are applicable, the method specialized for, e.g., two notes is run whenever two notes are supplied as argument, even though another method, specialized to two musical objects, is applicable as well: the `note` class is more specific than the `musical- object` class because the first inherits from the second. Thus in those cases the method specialized for notes effectivelly shadows the less specific one.[3]The auxillary functions that test for equality of lists and sets are written in true Common Lisp style, with an extra argument that will function as the equality test to be used for their items. Thus they thus embody only control structure of visitng elements in lists and sets while testing for equality, and they can be supplied with the same? predicate in case the test is for equality of lists or sets of musical objects, as is done above.

```
(defun same-list (list1 list2 &key (test #'eql))

"Do two lists contain the same items in the same order"

(loop for item1 in list1

for item2 in list2
```

```
always (funcall test item1 item2)))


(defun same-set (set1 set2 &key (test #'eql))

"Do two list contain the same items in any order"

(and (= (length set1)(length set2))

(loop for item1 in set1

always (member item1 set2 :test test))))
```

The `same?` predicate can be used as general equality test, allowing the many Common Lisp fuctions that can handle a test predicate to be applied directly to musical objects. In the example below we search for the place of one note in the elements slot of a musical object using the Common Lisp function `position`.

```
(position (note :duration 4 :pitch "G2")

(elements (example))

:test #'same?)

->

1
```

A window with a selection

Since we have created the possibility to do an operation on a musical object stored (and displayed) in a window, and we have written transformations that can operate on parts of a musical object, it seems a good idea to be able to select such a part and store it in a window, to be able to act on it without having to search for it every time. As a first solution we will add this behavior to draw-window: it will get an extra slot to store a selection, a part of its musical object, and some ways to update this selection. As acces functions to this facility we define a `select` method to store a specified part as selection, a `select- all` method which indicates that the whole musical object is selected, a `deselect` method to reset the selection to contain noting. Because we forsee that a change of the selected part may need to trigger some more actions, e.g., when we will have create a distict visible appearance of a selected object, all updates to the selction are channeled through a call to `select`. This narrow interface in than a nice hook to provide for future extensions. because in CLOS we cannot enforce the use of such an interface, and prohibit e.g. direct updates of the `selection` slot, a CLOS programmer needs some extra discipline to design and documents these design decisions. However, later we will see how in the meta-object protocol we can program such encapsulation ourselves. **[to do: meta object protocol object encapsulation interface]**

```
(defclass draw-window (piano-roll-contents-draw-window)

((selection :accessor selection :initarg :selection :initform nil))

(:documentation "Our new pianoroll window class"))


(defmethod select ((window draw-window) object)
```

```
"Select an object in the window"

(setf (selection window) object))


(defmethod select-all ((window draw-window))

"Select the whole musical object in the window"

(select window (object window)))


(defmethod deselect ((window draw-window))

"Set the window selection to nil"

(select window nil))
```

The next step is to add support for acting on selections. The `selection- operation` function closely resembles `window- operation`, but now the operation is don on the selection, not the whole musical object. It has to be taken into account that a selection may be empty, in that case no operation needs to be carried out.

```
(defun selection-operation (window operation &rest args)

"Apply the operation to a musical object stored in a window"

(when (and window (selection window))

(apply operation (selection window) args)

(invalidate-view window t))

window)
```

Adding some more auxilary functions will make it convenient to act on a selection in the front window.

```
(defun front-selection-operation (operation &rest args)

"Apply an operation to the selection stored in the frontmost draw window"

(apply #'selection-operation (front-draw-window)

operation

args))


(defmethod front-select-all ()

"Select the complete musical object in the front window"

(do-front-window #'select-all))
```

```
(defmethod front-deselect ()
"Deselect everything in the front window"
(do-front-window #'deselect))

(defmethod front-select (selector &rest args)
"Apply a function which will state what to select in the front window"
(do-front-window
#'(lambda(window)
(select window
(apply #'find-musical-object-onset (object window) 0 selector args)))))
```

[to do: ; eruit ? gebruikt allen voor debug]

```
(defmethod front-selection ()
"Return the selection of the front window"
(do-front-window #'selection))
```

These facilities can now be tested:

```
(draw (example)) => 16.A. draw the example
(front-select-all)
(front-selection-operation #'transpose 3) => 16.B. transposition of the full selection
(front-select (compose-and (use-first-arg #'sequential?)
(use-first-arg #'identity)))
(front-selection-operation #'mirror 60) => 16.C. reflect the sequential part
```

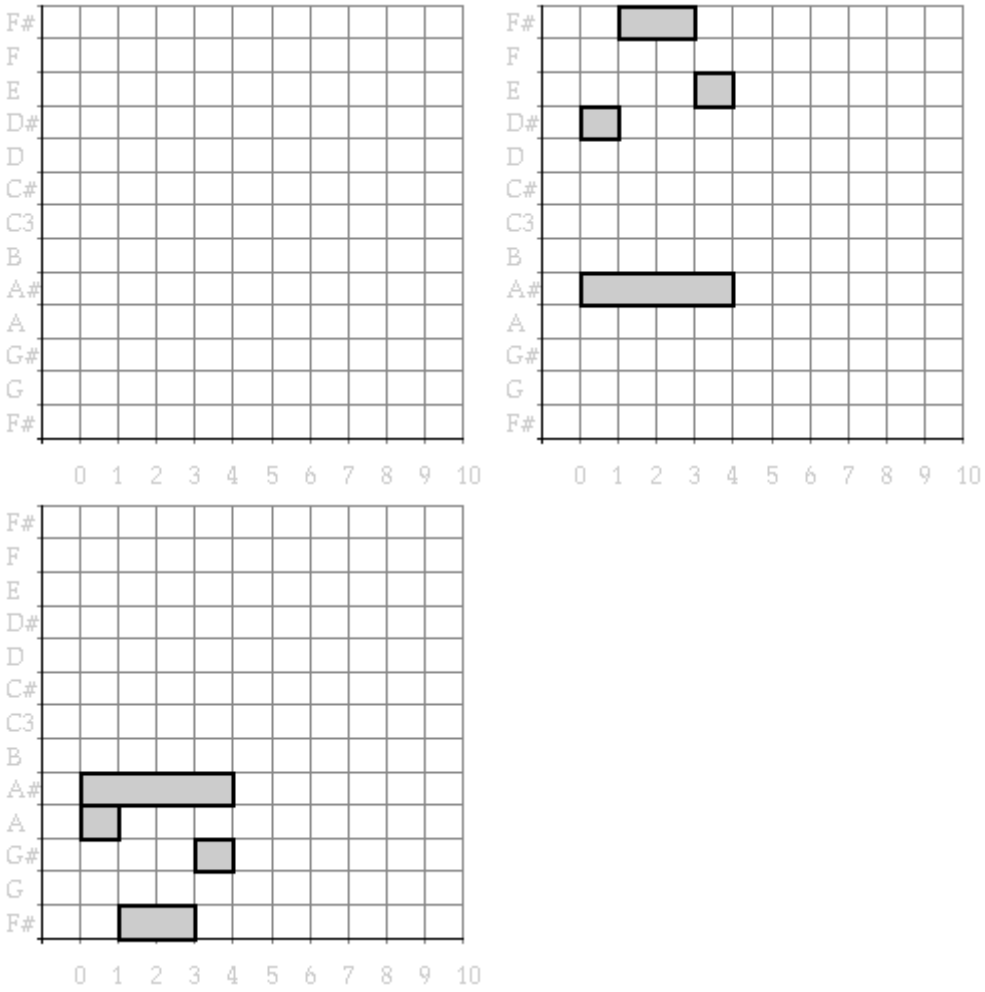

Figure 16. testing operations on a selection

[to do: boom op de kop maken voor een soort boom]

[to do: de aanroep hierachie boom hier zetten]

Multiple inheritance

Though at first sight everything seems fine, our modularity has suffered from adding these new facilities: any draw window now has the added selection behaviour and our virgin draw window is gone forever. It would have been much better had we defined a new window class to contain this behavior. Let us separate the new facilities from the main `draw- window` class.

```
(defclass selection-mixin ()

((selection :accessor selection :initarg :selection :initform nil))

(:documentation "A window mixin to add selection behavior"))
```

```
(defmethod select ((window selection-mixin) object)

"Select an object in the window"

(setf (selection window) object))


(defmethod select-all ((window selection-mixin))

"Select the whole musical object in the window"

(select window (object window)))


(defmethod deselect ((window selection-mixin))

"Set the window selection to nil"

(select window nil))
```

To create a new `draw- window` class which supports a selection we now have to inherit from both the `piano- roll- contents- draw- window` class and the `selection- mixin` class.

```
(defclass draw-window (selection-mixin piano-roll-contents-draw-window)

()

(:documentation "Our current pianoroll window class"))
```

This is the first time we make use of so-called *multiple inheritance*. This powerfull construct is not allowed in many other object oriented languages and, one has to acknowledge, it can be misused easily. The programmer has to exercise discipline in its use to keep the complexity managable. Some discipline was already used above: the reader may have noticed that the selecition mixin class does not inherit from any class, in particulary not from the piano roll or any other window class. In that sense it is not a complete class, it can not be instantiatiated to yield anything usefull. This so called *abstract class* or *mixin class* is only intended to supply a certain chunck of behavior that is to be mixed in with another class to be meaningfull. Because CLOS has no language constructs to specify that our mixin cannot be instatiated on its own[4], part of the programmers discipline is to reflect this nature in the name of the class. Of course the selection window class could have inherited from the piano roll window class. It would work, and all inheritance issues would still be well defined in CLOS, but since then the same methods and slots can be inherited via multiple routes, keeping track conceptually of the design of our system becomes a much more complex issue. [5]:footnote Again, in anticipation, we generalize more then actually needed at the moment of implementing a certain solution.

```
(defclass draw-window-mixin ()()

(:documentation "Additional behavior for draw window"))
```

```
(defclass selection-mixin (draw-window-mixin)

((selection :accessor selection :initarg :selection :initform nil))

(:documentation "Our pianoroll window class"))
```

[to do: ? tree figure of multiple inheritance (with abstract class)]

Binding our own programs to existing menu commands

The `select- all` operation may have looked familiar to you because it is defined in many other Macintosh applications as well. And part of the quality of the design of the Machintosh software lies in the ease which which users can transfer knowledge they learned in one application to other programs. In that same line it is wise to recognize that our select-all is conceptually the same as select-all commands for other applications on other windows, and that its invocation should be supported by the same means. Surprisingly this has already been taken care of: pull down the file menu while a draw window is selected. The select all command will not be grey: because we defined a `select- all` method applicable to this window, it is made accessible as menu action and given the **[to do: clover invoegen]**clover-A command key automaticaly. Try it, and think of what other standard Machintosh operations would be usefull to support.

Binding commands to our own menu and control keys

There are of course commands specific to musical objects like `play` and `transpose`, which cannot find a place in the system menus. We will create a new menu for these types of commands, define a command key for them and let them all operate on the selection made in the front window. The commands to be put in the menu and bound to command keys will, for now, have no arguments other than the window whose selection they have to be applied to.

```
(defmethod play-selection ((window draw-window))

"Play the musical object selected in the window"

(selection-operation window #'play))
```

```
(defmethod transpose-selection-up ((window draw-window))

"Transpose the musical object selected in the window up by one semitone"

(selection-operation window #'transpose 1))
```

```
(defmethod transpose-selection-down ((window draw-window))

"Transpose the musical object selected in the window down by one semitone"

(selection-operation window #'transpose -1))
```

```
(defmethod mirror-selection ((window draw-window))

"Reflect the musical object selected in the window around Middle C"
```

```
(selection-operation window #'mirror 60))


(defmethod retrograde-selection ((window draw-window))
"Reverse the musical object selected in the window"
(selection-operation window #'retrograde))


(add-music-command "Play" #'play-selection #\p)
(add-music-command "Transpose up" #'transpose-selection-up #\u)
(add-music-command "Transpose down" #'transpose-selection-down #\d)
(add-music-command "Mirror" #'mirror-selection #\m)
(add-music-command "Retrograde" #'retrograde-selection #\r)


(draw (example))
(front-select-all) => 20.A. select the whole example
(front-draw-window) => 20.B. choose the retrograde command in the menu
(front-selection-operation #'Retrograde) => 20.C. the result
```

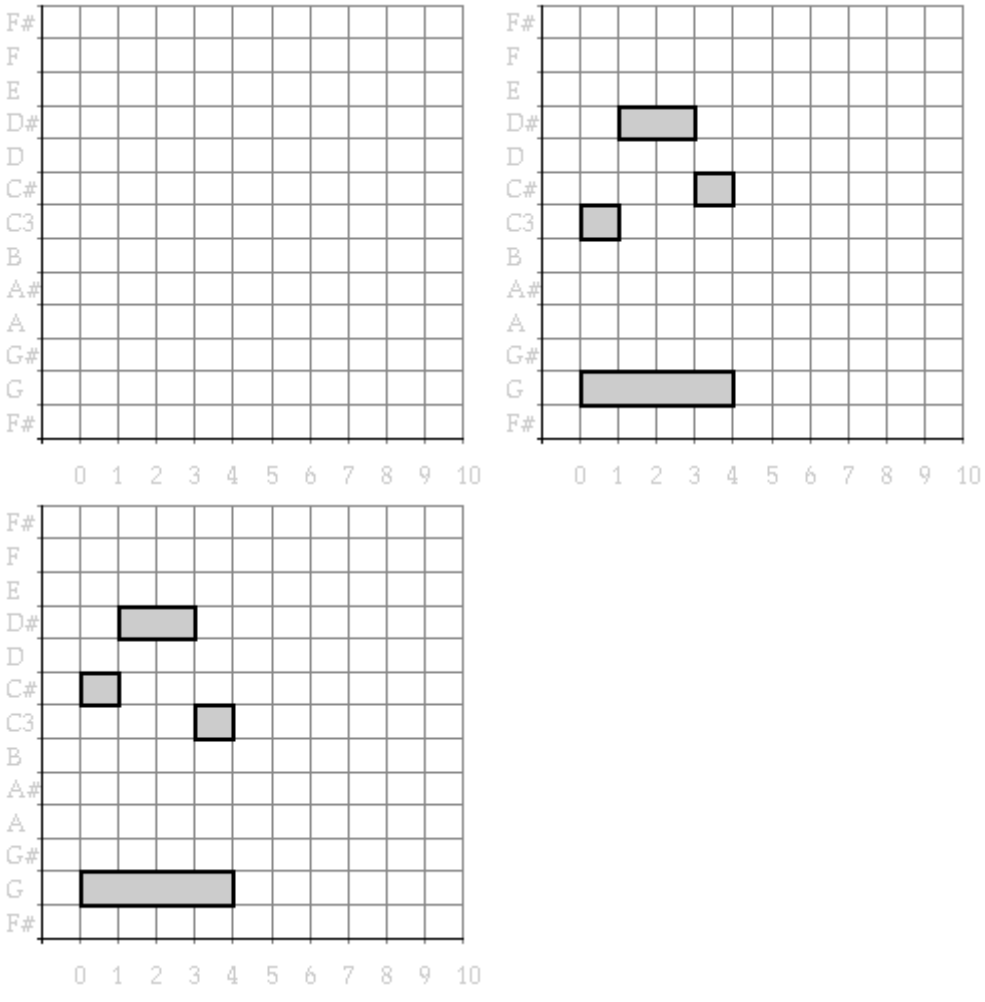

Figure 20. testing operations on a selection

[to do: plaatje van het music menu]

multi-methods, method combination and multiple inheritance combined

The reader might have noticed that the addition of shading to the drawing of notes is hard-wired into our draw-window, it cannot be switch off anymore. Were we to add the possibility of both shaded and unshaded drawing to our program in a traditional way, we would add an extra parameter (a flag), and consult it in the drawing of each note to check whether shading is desired. This would make the program bulky and less modular, especially when more drawing options would arise (and more parameters need to be added). A better, and thoroughly object oriented solution is based on the realisation that there are different classes of windows, each with its own 'look'. This entails that drawing behavior does not only depend on the kind of musical object, but on the type of window as well, which is just the kind of issue that multi methods can address elegantly. Thus shading could have been formalised as a window mixin class as well, instead of an addition to the draw window itself, and indeed it should have. The shading takes place in the `draw- boundary` :before method which takes a musical object, a boundary and a window. This multi-method only needs to be redefined for a new `shaded- mixin` class, instead of for all draw windows.

```
(defclass shaded-mixin (draw-window-mixin)

()

(:documentation "A window that shades the notes of the pianoroll"))


(fmakunbound 'draw-boundary)


(defmethod draw-boundary :before ((object note) boundary (window shaded-mixin))

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window boundary :light-gray))


(defmethod draw-boundary ((object note) boundary (window draw-window))

"Draw a box with a specified boundary (left top right bottom)"

(draw-rectangle window boundary))


(defclass draw-window (shaded-mixin

selection-mixin
```

```
piano-roll-contents-draw-window)

()

(:documentation "Our (forever changing) pianoroll window class"))
```

In the example above we thus have used three main constructs of CLOS which form such a potent mixture in combination: Multiple inheritance (to have the `draw- window` class inherit from more classes, each supplying part of the behavior needed), multi-methods (to select the `draw- boundary` method on the basis of both the musical object and the window class) and declarative method combination (to supply a piece of code that has to be run before the main drawing of a boundary). Not only is the extension of our drawing program implemented completely as an addition of a small piece of code (a class and a method), but that also old behavior is not affected (old fashioned draw-windows without shading can still be build and will function in the familiar way.

Showing structure

Sequential and parallel structure is paramount in our examples, but it is not visible in the piano roll graphics. To make up for that omission we can visualize the structure of a musical object in the form of a box extending for the duration of the whole object and covering its pitch range. The first idea would be to add a before or after method to draw-musical-object for compound musical objects, to calculate the boundary and draw a box around them. This may later give rise to inefficencies, since new draw mixins may need the boundary of the compound objects as well - just like the before method in the shade mixin did for notes. We can best admit defeat, our basic draw window did not supply enough hooks in the drawing code to make it reusable enough. [6] Luckily it is fun to try to evolve the base class definitions such that many extensions can be hooked unto it. So that is what we need to do here: add a call to `draw-boundary` in the drawing of compound musical objects. And add stub methods that do nothing to calulate and draw boundaries for objects other then notes. Thus a control structure is errected in which every object is visited with a call to `draw- boundary`, which could trigger the drawing of a musical object with a given boundary, possibly implemented in a window mixin. And the calculations of the boundaries themselves are to be implemented as a method for boundary, also possibly implemented in a mixin class. Our rewrite is shown below.

```
(fmakunbound 'draw-musical-object)


(defmethod draw-musical-object ((object compound-musical-object)

onset

(window draw-window))

"Draw a graphical piano-roll representation of a compound musical object"

(draw-boundary object (boundary object onset window) window)

(mapc-elements-onsets object onset #'draw-musical-object window))


(defmethod draw-musical-object ((object basic-musical-object)

onset
```

```
(window draw-window))

"Draw a graphical representation of a basic musical object"

(draw-boundary object (boundary object onset window) window))


(defmethod draw-boundary ((object note) boundary (window draw-window))

"Draw a note with a specified boundary (left top right bottom)"

(draw-rectangle window boundary))


(defmethod draw-boundary ((object musical-object) boundary (window draw-window))

"Draw nothing"

(declare (ignore boundary))

)


(defmethod boundary ((object note) onset (window draw-window))

"Return a list of the sides of a graphical representation of a note"

(declare (ignore window))

(list onset ; left

(+ (pitch object) .5) ; top

(offset object onset) ; right

(- (pitch object) .5))) ; bottom


(defmethod boundary ((object musical-object) onset (window draw-window))

"Return nil, boundary of musical object is not known"

(declare (ignore onset))

nil)
```

The above modularity has the added advantage that the calulation of boundaries are separated from the drawing based on them. This allows, e.g., for mixing in a class to prevent the drawing of objects which lay completely outside the visible part of the window, which needs the calculation of boundaries of compound musical object but not nessary the drawing of them. But let's get back on track and design how to draw contours around sequential and parallel object, and how to calculate these countours or boundaries. The boundary method for compound musical objects has to recursively call itself on each element and combine the results by calculating the highest, rightmost, lowest and leftmost of the sides of the boundaries of its elements. Just like a graphical representation cannot easily be invented for rests, it is not easy to define boundaries for compound objects containing only rests. So we will not calculate those in the hope that later better ideas will arise.

```
(defclass combine-boundary-mixin (draw-window-mixin)

()

(:documentation "A mixin to combine boundary boxes"))


(defmethod boundary ((object compound-musical-object) onset (window combine-boundary-
mixin))

"Return a list of the sides of a box enclosing a compound musical object"

(apply #'combine-boundaries

(map-elements-onsets object onset #'boundary window)))


(defmethod combine-boundaries (&rest boundaries)

"Combine a list of boundary boxes into one box that encloses them all"

(let ((true-boundaries (remove nil boundaries)))

(when true-boundaries

(loop for (x1 y1 x2 y2) in true-boundaries

minimize x1 into left

maximize y1 into top

maximize x2 into right

minimize y2 into bottom

finally (return (list left top right bottom))))))
```

The minimization and maximization of times and pitches can be done easily using the loop macro facilities. Since the `boundary` method may return nil for certain kinds of objects (like rests), we have to carefully remove these before the list of boundaries is combined into one. [7]

Now we can define the class which actually draws the boundaries. It inherits from `combine- boundary- mixin` because it critically depends on the calculation of boundaries defined in that class. This makes the listing of the `combine- boundary- mixin` class in the list of classes of which the `draw- window` inherits superfluous: all inheritance of methods of the `combine- boundary- mixin` is taken care of via `structured- mixin`. For simplicity we ignore for the moment that one might want to properly nest the boxes graphically - enclosing boxes will be drawn over each other. The extension only amounts to a redefintion of the `draw- boundary` method that may now be applied to any musical object. We only have to prevent drawing when it is called with a `nil` boundary.

```
(defclass structured-mixin (combine-boundary-mixin)

()
```

```
(:documentation "A mixin to visualize internal structure"))


(fmakunbound 'draw-boundary)


(defmethod draw-boundary ((object musical-object) boundary (window structured-mixin))
"Draw a box with a specified boundary (left top right bottom)"
(when boundary
(draw-rectangle window boundary)))


(defclass draw-window (structured-mixin
shaded-mixin
selection-mixin
piano-roll-contents-draw-window)
()
(:documentation "The flexible pianoroll window class"))


(draw (repeat (example))) =>
```

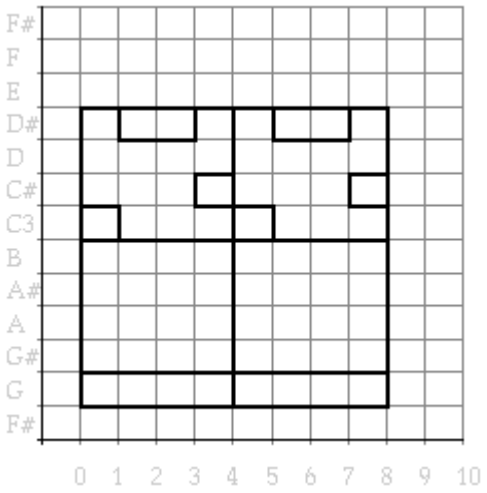

Figure 21. A musical object with its structure visualized.

The call hierarchy is given in Figure 22. We will show later how this structure can be simplified a bit.

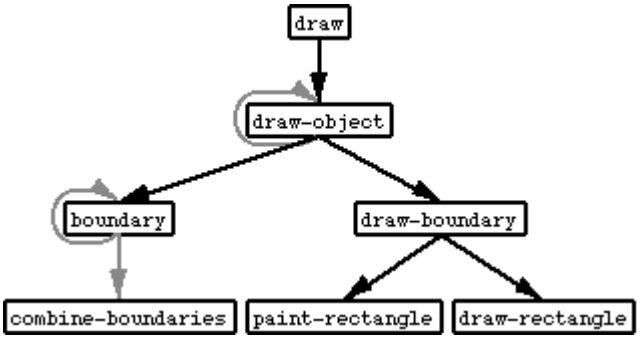

Figure 22. The complex flow of control while drawing.

[to do: check of dit klopt]

Making selections visible, around method

For our window that supports a selection now the rule that in a graphical user interface all internal state that affects the user actions should be made visible, needs to be applied: we need to show a selected (sub)object in a distinguised fashion. As always we will supply the extra behavior in the form of a window mixin. There are several possibilities to visualize a selection. We will opt here for giving the object a grey background. Thus the fact that we made the shading of notes a mixin already pays-off, we can now easily remove it from the list of classes that draw-window inherits from, and use shading for a different purpose. To make the selection appear consistent, we need to add an after select method which informs the window system that a redraw is needed. Because all updates to the selection are made via `select`, this good disipline pays of here: we can be sure that this is the only place that needs patching.

```
(defclass visible-selection-mixin (selection-mixin)

()

(:documentation "A class to support visible selections in draw windows"))


(defmethod draw-boundary :before ((object musical-object)

boundary

(window selection-mixin))

"Paint selected objects"

(when (eql object (selection window))

(paint-rectangle window boundary :light-gray)))


(defmethod select :after ((window selection-mixin) object)
```

```
(invalidate-view window t))


(defclass draw-window (structured-mixin
visible-selection-mixin
selection-mixin
piano-roll-contents-draw-window)
()
(:documentation "The pianoroll window class, selection-controlé"))
```

Note how the select after method is not specialized to musical objects: it may have to be applied to nil.

```
(draw (example)) => 26.A. draw the example
(front-select-all) => 26.B. selection of whole musical object
(front-select (compose-and (use-first-arg #'sequential?)
(use-first-arg #'identity))) => 26.C. selection of the sequential part
```

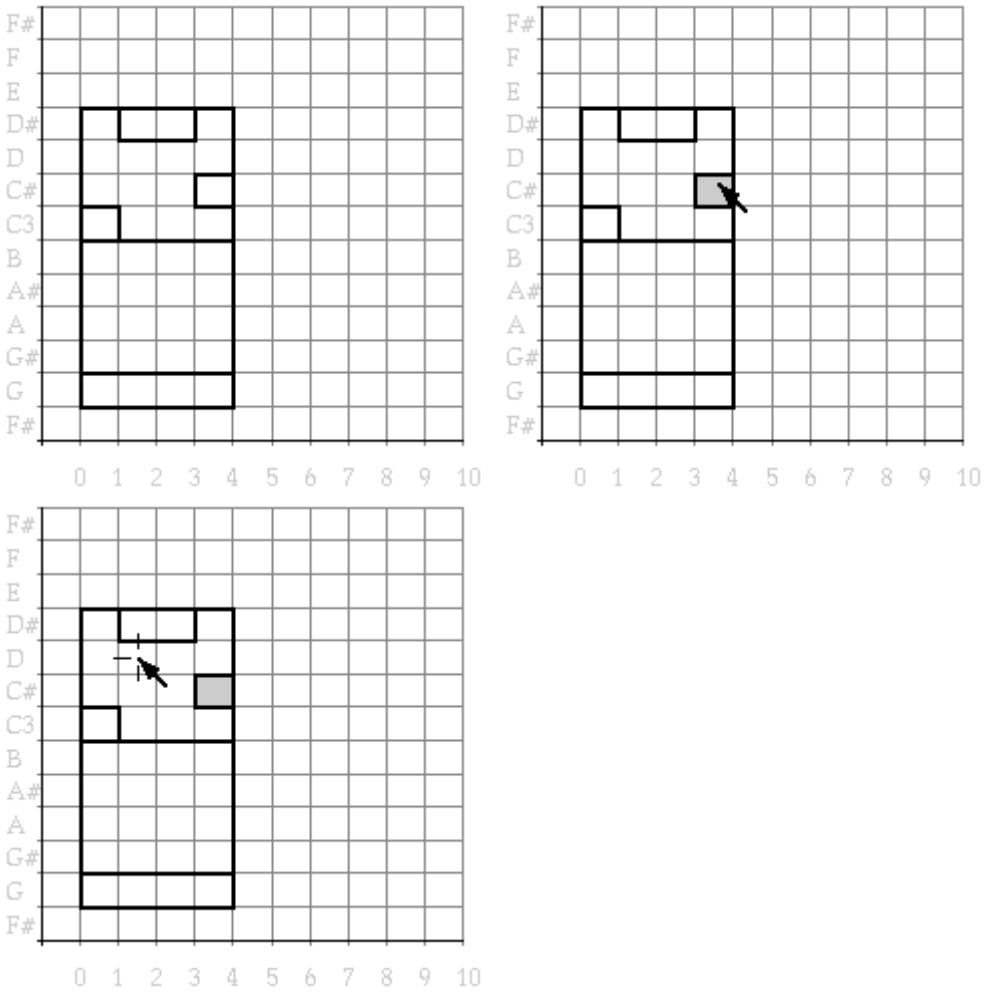

Figure 26. testing operations on a selection

Making musical objects mouse sensitive

You knew this had to come sooner or later didn't you: we want to select musical objects with the mouse. Again there are several possibilities to define the area that will select a compound musical object, one of them is the boundary itself. The other possibility, that we will explore here interprete mouse-clicks in the smallest boundary thta contains it. Thus we first have to look for notes, and whether the mous was on one of them, then we go one level up to check whether the mouse was in the boundary box of a compound musical object (but not on its notes), etc. The flow of control is thus exactly oppposite the top-down way in which `find- musical- object- onset` searches through the structure. By reversing the order of the clauses in the `or` in the body of that function, our bottom-up search can be realized with one simple change.

```
(defmethod deep-find-musical-object-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing its sub-objects, then itself"

(or (apply #'find-element-onset object onset

#'deep-find-musical-object-onset predicate args)

(apply predicate object onset args)))


(defmethod deep-find-musical-object-onset ((object basic-musical-object)

onset predicate &rest args)

"Return predicate's result, when satisfied for this object at its onset"

(apply predicate object onset args))


(defmethod covers? ((object musical-object) onset window time pitch)

"Does the point (time, pitch) lay in the boundary of the musical object?"

(destructuring-bind (onset max-pitch offset min-pitch)

(boundary object onset window)

(and (<= onset time offset)

(<= min-pitch pitch max-pitch)

object)))


(defmethod select-cover ((window window) time pitch)
```

```
"Select the musical object at (time, pitch) in the window"

(select window

(deep-find-musical-object-onset (object window) 0 #'covers?

window time pitch)))
```

We can check our search by hand before we have access to the mouse sensitivity.

```
(draw (example))

(do-front-window #'select-cover .5 60) =>
```

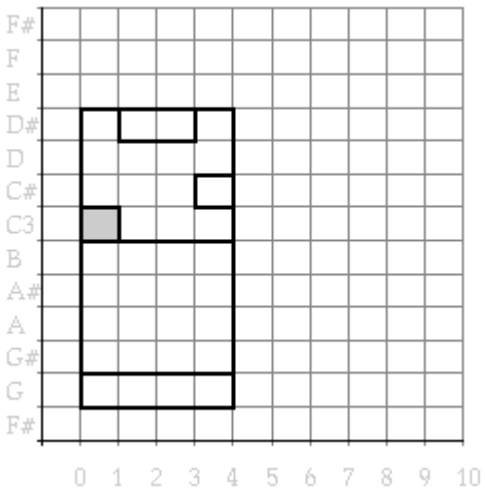

Figure 27. Testing the selection by pointing to a position

We rely on the piano roll window to translate the screen coordinates to our time-pitch domain because that is the proper place for that knowledge to reside, (think, e.g., of a later adition of a scroll bar in the window, which effectively modifies the screen to domain mapping). [8] The piano roll window window will send a `mouse- click` message (with the moused time and pitch positions, the modifier keys and the number of clicks) to itself that we catch and act upon by selecting the smallest sub-object that covers mouse. An `eql` specialisation on the last two arguments ensures that only single mouse clicks without modifier keys are being acted upon.

```
(defclass mouse-select-mixin (selection-mixin)

()

(:documentation "A mixin that makes musical objects mouse selectable"))
```

```
(defmethod find-position-in-musical-object ((object musical-object)
(window mouse-select-mixin) time pitch)
"Return the (sub)object at (time, pitch)"
(find-musical-object-onset object 0 #'covers? time pitch))

(defmethod mouse-click ((window mouse-select-mixin)
time pitch (modifiers (eql nil)) (count (eql 1)))
"Select in response to a single mouse click at (time, pitch) without modifier keys"
(select-cover window time pitch))

(defclass draw-window (mouse-select-mixin
structured-mixin
visible-selection-mixin
piano-roll-contents-draw-window)
()
(:documentation "A mouse-friendly pianoroll window class"))
```

Now the fun can really start: mouse-selecting objects, applying transformations on the by using the menu or command keys: our basic user interface is in place.

```
(defmethod front-mouse-click (&rest args)
(apply #'do-front-window #'mouse-click args))

(draw (example)) => 29.A. click on an object
(front-mouse-click 3.6 61.2 nil 1) => 29.B. selected object
(front-draw-window) => 29.C. click elsewhere
(front-mouse-click 1.5 62 nil 1) => 29.D. new selection
```

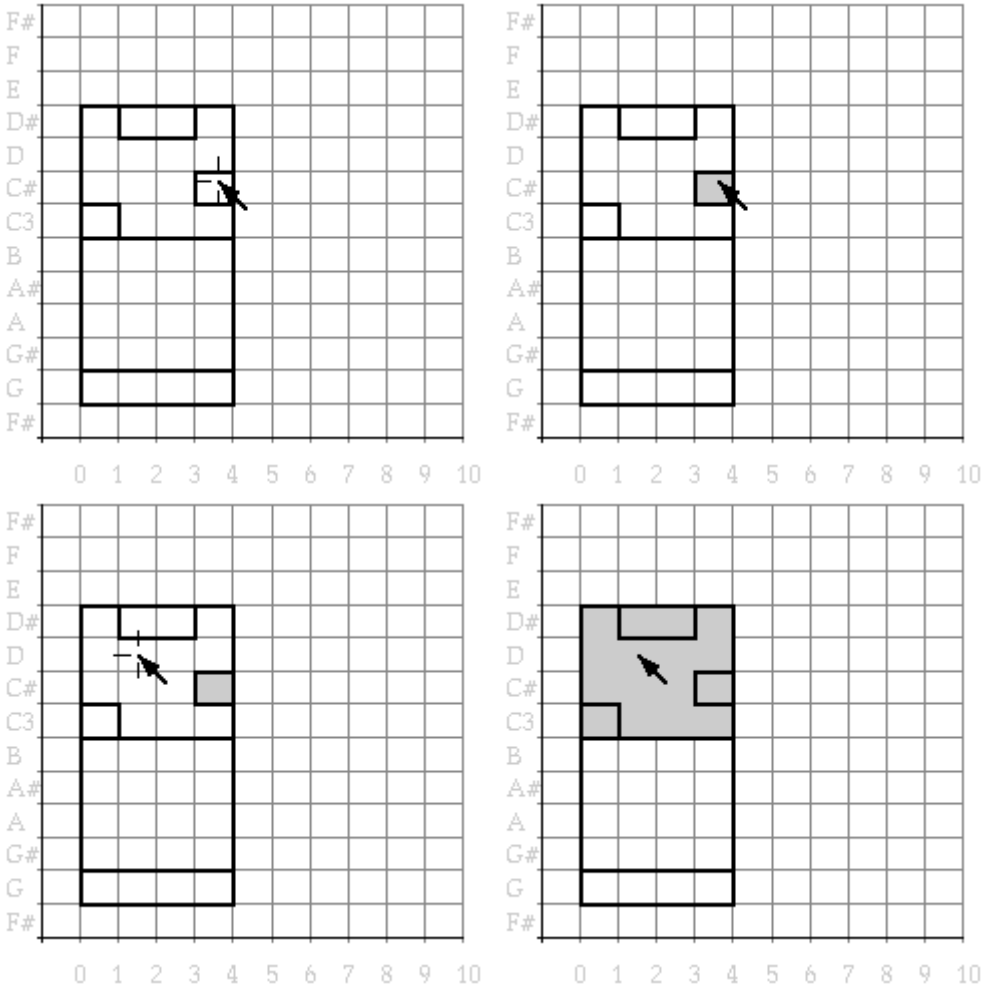

Figure 29. Selections in the example made visible

[to do: waar geven we commando's (bv transpose) argumenten pas in embedded of al eerder, hier?]

[to do: backpointers maken canon en repeat mogenlijk als destructieve window operatie die toevoegen in menu etc.]

Definitions made

`view-draw-contents (window)` *`method`*

(Re)draw the stored object as a pianoroll

`visible-selection-mixin (selection-mixin)` *`class`*

A class to support visible selections in draw windows

`mouse-select-mixin (selection-mixin)` *`class`*

A mixin that makes musical objects mouse selectable

`combine-boundary-mixin (draw-window-mixin)` *`class`*

A mixin to combine boundary boxes

`structured-mixin (combine-boundary-mixin) class`

A mixin to visualize internal structure

`draw-window (mouse-select-mixin structured-mixin visible-selection-mixin piano-roll-contents-draw-window) class`

A mouse-friendly pianoroll window class

`piano-roll-contents-draw-window (piano-roll-window) class`

A window for drawing and re-drawing musical objects

`shaded-mixin (draw-window-mixin) class`

A window that shades the notes of the pianoroll

`draw-window-mixin nil class`

Additional behavior for draw window

`map-time (object onset time-map) method`

Adjust the duration of a basic musical object according to the time map

`front-select (selector &rest args) method`

Apply a function which will state what to select in the front window

`front-window-operation (operation &rest args) function`

Apply an operation to a musical object stored in the frontmost draw window

`front-selection-operation (operation &rest args) function`

Apply an operation to the selection stored in the frontmost draw window

`selection-operation (window operation &rest args) function`

Apply the operation to a musical object stored in a window

`window-operation (window operation &rest args) function`

Apply the operation to a musical object stored in a window

`combine-boundaries (&rest boundaries) method`

Combine a list of boundary boxes into one box that encloses them all

`front-deselect nil method`

Deselect everything in the front window

`same-set (set1 set2 &key (test #'eql)) function`

Do two list contain the same items in any order

`same-list (list1 list2 &key (test #'eql)) function`

Do two lists contain the same items in the same order

`covers? (object onset window time pitch) method`

Does the point (time, pitch) lay in the boundary of the musical object?

`draw (object &rest args &key window-class &allow-other-keys) method`

Draw a graphical piano-roll representation of a compound musical object

`draw-musical-object (object onset window) method`

Draw a graphical representation of a basic musical object

`selection-mixin (draw-window-mixin) class`

Our pianoroll window class

`draw-boundary (object boundary window) method`

Paint selected objects

`play-selection (window) method`

Play the musical object selected in the window

`mirror-selection (window) method`

Reflect the musical object selected in the window around Middle C

`sounding-pitches (object onset from to) method`

Return a list of the pitch of the note, when sounding in interval [from, to]

`boundary (object onset window) method`

Return a list of the sides of a box enclosing a compound musical object

`make-tempo-change (duration begin end) function`

Return a time-warp function based on begin and end tempo factors

`find-element-onset (object onset predicate &rest args) method`

Return first non-nil predicate's result, testing elements and their onset

`same? (object1 object2) method`

Return nil as default case: two musical objects of different type are unequal

`deep-find-musical-object-onset (object onset predicate &rest args) method`

Return predicate's result, when satisfied for this object at its onset

`find-musical-object-onset (object onset predicate &rest args) method`

Return predicate's result, when satisfied for this object at its onset

`find-position-in-musical-object (object window time pitch) method`

Return the (sub)object at (time, pitch)

`collect-sounding-pitches (object onset from to) method`

Return the empty list representing silence

`front-draw-window nil` *function*

Return the frontmost drawing window, bringing it to the front as well

`map-elements-onsets (object onset operation &rest args)` *method*

Return the results of the operation applied to each element and its onset

`front-selection nil` *method*

Return the selection of the front window

`square (x)` *function*

Return the square of a number

`set-union (&rest args)` *function*

Return the union of any number of sets

`retrograde (object)` *method*

Reverse all components of the object in time

`retrograde-selection (window)` *method*

Reverse the musical object selected in the window

`mouse-click (window time pitch modifiers count)` *method*

Select in response to a single mouse click at (time, pitch) without modifier keys

`front-select-all nil` *method*

Select the complete musical object in the front window

`select-cover (window time pitch)` *method*

Select the musical object at (time, pitch) in the window

`select-all (window)` *method*

Select the whole musical object in the window

`deselect (window)` *method*

Set the window selection to nil

`transpose-selection-down (window)` *method*

Transpose the musical object selected in the window down by one semitone

`transpose-selection-up (window)` *method*

Transpose the musical object selected in the window up by one semitone

`use-first-arg (function)` *function*

Wrap the function such that it can be given spurious arguments

`front-mouse-click (&rest args)` *method*

`select (window object)` *method*

`do-front-window (action &rest args)` *function*

Literature references made

Balaban, M. 1992 Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music. in Balaban, M., K. Ebcioglu & O. Laske (Eds.) Cambridge, Mass.: MIT Press. 111-138

Chowning, 1973 Journal of the Audio Engineering Society, 21(7):526-534. Reprinted inC. Roads and J. Strawn, (Eds.) 1985 Foundations of Computer Music. Cambride, MA: MIT Press. pp. 6-29.

Dannenberg, R.B. 1991 Real-time Scheduling and Computer Accompaniment. in Mathews, M.V and J.R. Pierce (Eds.) . Cambridge, MA: MIT Press.

Dannenberg, R.B. 1993 The Implementation of Nyquist, a Sound Synthesis Language. in San Francisco: ICMA.

Dannenberg, R.B., P. Desain and H. Honing (in Press) Programming Language Design for Music. in G. De Poli, A. Picialli, S.T. Pope & C. Roads (Eds.) Lisse: Swets & Zeitlinger.

Desain, P. and H. Honing (In Preparation) CLOSe to the edge, Advanced Object-Oriented Techniques in the Representation of Musical Knowledge. Jounral of New Music Research.

Desain, P. 1990 Lisp as a second language, functional aspects. (28)1 192-222

International MIDI Association 1983 North Holywood: IMA.

Moore, F.R. 1990 Englewood Cliffs, New Jersey: Prentice Hall.

Pope, S.T. 1992 The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers.16(3) 73-91

Pope, S.T. 1993 Machine Tongues XV: Three Packages for Software Sound Synthesis. 17(2) 23-54

Steele, G.R. 1990 Bedford MA: Digital Press.

Tatar,D.G. 1987 Bedford, MA: Digital Press.

Glossary references made

abstract class

abstract class

access function

all functions part of a data abstraction layer (selector and constructor functions). [loaded from Glossary Functional]

accessor function

after method

anonymous function

A function whose ‘pure’ definition is given, not assigning it a name at the same time. [loaded from Glossary Functional]

applicable method

applicable methods

application

obtaining a result by supplying a function with suitable arguments. [loaded from Glossary Functional]

assignment

atom

in Lisp: any symbol, number or other non-list. [loaded from Glossary Functional]

before method

class

class hierarchy

class options

combinator

A function that has only functions as arguments and returns a function as result. [loaded from Glossary Functional]

compile time

congruent lambda lists

cons

A Lisp primitive that builds lists. Sometimes used as verb: to add an element to a list. [loaded from Glossary Functional]

constant function

A function that always returns the same value [loaded from Glossary Functional]

constructor function

A function that as part of the data abstraction layer provides a way of building a data structure from its components. [loaded from Glossary Functional]

constructor functions

continuations

A way of specifying what a function should do with its arguments. [loaded from Glossary Functional]

coroutines

parts of the program that run in alternation, but remember their own state of computation in between switches. [loaded from Glossary Functional]

data abstraction

A way of restricting access and hiding detail of data structures [loaded from Glossary Functional]

data abstraction

data type

A class of similar data objects, together with their access functions. [loaded from Glossary Functional]

declaration

declarative

declarative method combination

dialect

A programming language can be split up into dialects that only differ (one hopes) in minor details. Lisp dialects are abundant and may differ a lot from each other even in essential constructs. [loaded from Glossary Functional]

direct superclasses

dynamically typed language

effective method

first class citizens

rule by which any type of object is allowed in any type of programming construct. [loaded from Glossary Functional]

free variables

function

A program or procedure that has no side effects [loaded from Glossary Functional]

function composition

the process of applying one function after another. [loaded from Glossary Functional]

function quote

A construct to capture the correct intended meaning (with respect tothe current lexical environment) of a anonymous function so it can beapplied later in another environment. It is considered good programming style to use function quotes as well when quoting the name of a function. [loaded from Glossary Functional]

functional abstraction (or procedural abstraction)

A way of making a piece of code more general by turning part of it into a parameter, creating a function that can be called with a variety of values for this parameter. [loaded from Glossary Functional]

functional argument (funarg)

A function that is passed as argument to another one (downward funarg) or returned as result from other one (upward funarg) [loaded from Glossary Functional]

generalizedvariable

generic

global variables

an object that can be referred to (inspected, changed) from any part of the program. [loaded from Glossary Functional]

higher order function

A function that has functions as arguments. [loaded from Glossary Functional]

imperative style

inheritance

initializationkeyword

initializationprotocol

initialization argument list

initialization keyword

instance

instantiation

iteration

repeating a certain segment of the program. [loaded from Glossary Functional]

lambda list keyword

lambda-list keyword

A keyword that may occur in the list of parameter names in a function definition. It signals how this function expects its parameters to be passed, if they may be ommited in the call etc. [loaded from Glossary Functional]

lexical scoping

A rule that limits the 'visibility' of a variable to a textual chunk of the program. Much confusion can result from the older- so called dynamic scoping - rules. [loaded from Glossary Functional]

list

message passing

message passing style

meta object protocol

method

method combination

method combination

The declaritive way in which CLOS allows more methods to be bundled and run, in situations where more are applicable [loaded from Object Oriented I]

method qualifier

methods

mixin class

modification macro's

most specific method

multi-methods

multi-methods

multiple inheritance

multiple inheritance

name clash

name clashes

object oriented programming

A style of programming whereby each data type is grouped with its own access function definitions, possibly inheriting them from other types. [loaded from Glossary Functional]

object-oriented style

parameter-free programming

A style of programming whereby only combinators are used to build complex functions from simple ones. [loaded from Glossary Functional]

part-of hierarchy

polymorphic

polymorphism

prefix notation

A way of notating function application by prefixing the arguments with the function. [loaded from Glossary Functional]

primary method

primary methods

procedural

quote

A construct to prevent the Lisp interpreter from evaluating an expression. [loaded from Glossary Functional]

read-eval-print

record

recursion

A method by which a function is allowed to use its own definition. [loaded from Glossary Functional]

run time

selector function

A function that as part of the data abstraction layer provides access to a data structure by returning part of it. [loaded from Glossary Functional]

selector functions

setf method

shadow

side effect

Any actions of a program that may change the environment and so change the behavior of other programs. [loaded from Glossary Functional]

side effect

slot descriptors

slots

stack

A list of function calls that are initiated but have not yet returned a value. [loaded from Glossary Functional]

standard method combination

statically typed

structure preserving

A way of modifying data that keeps the internal construction intact but may change attributes attached to the structure. [loaded from Glossary Functional]

tail recursion

A way of recursion in which the recursive call is the ‘last’ thing the program does. [loaded from Glossary Functional]

the most specific method

untyped language

[an error occurred while processing this directive]

[an error occurred while processing this directive]

Lisp as a second language, composing programs and music.

Peter Desain and Henkjan Honing

Chapter III Object-Oriented Style III

Draft mars 11, 1997

1. From iterator to recursor again
2. And again from iterator to recursor
3. event lists
4. Promoting event lists to musical objects
5. An around method can prevent a call of the primary method
6. In a primary method the next method may be called as well, but please don't
7. An around method can change the arguments of its primary method
8. An around method can modify the results returned by the primary method
9. Naming musical objects
10. initialize instance
11. customizing initialization
12. Setf methods
13. Arguments to user commands
14. Integer note maken en naar boven schuiven, na mixin, na after around, na after init, voor change window class
15. Meta Object Protocol: introspection allows objects to know their class
16. Meta Object Protocol: mixing class cocktails
17. Adding behavior to class definitions
18. Eql specialization, moet voor mouse-click
19. mop one/few instance classes
20. Standard method combination recap
21. build in and/append method combination
22. append build in method combination, na mop class-of
23. defined method combination
24. Loudness mixin voor note waar ??
25. Backpointers
26. maintaining back-pointer consistency
27. Destructive operations on the stucture of displayed musical objects
28. A clipboard for musical objects
29. Standard edit operations on window with musical objects
30. Open musical objects
31. draw rests

32. Score system
33. Sound
34. Mop protocol
35. mop cocktail methods
36. init keyword class adder
37. one instance class
38. Full code listing
39. Conclusion
40. Definitions made
41. Literature references made
42. Glossary references made

From iterator to recursor again

Remember the `find- musical- object- onset` recursor, which walks throuh a complete musical structure, uses `find- element- onset` to move in the width direction (through the elements list of an compound musical object), and supplies itself to this iterator to have itself applied recusively to proceed in the depth direction. In the same way we can use `mapc- elements- onsets` as basic horizontal iterator to create a true recursor to apply some operation to each (sub) object of a musical structure. But instead of a predicate to test, this recursor expects an operation to apply. [1]

```
(defmethod mapc-musical-object-onset ((object compound-musical-object)

onset operation &rest args)

"Apply the operation to this object, and to each sub-..-sub-object and their onsets"

(apply operation object onset args)

(apply #'mapc-element-onset object onset #'mapc-musical-object-onset operation args))
```

```
(defmethod mapc-musical-object-onset ((object basic-musical-object)

onset operation &rest args)

"Apply the operation to this object and its onset"

(apply operation object onset args))
```

Betere uitleg?, beginnen met basic, dan eerst uitschrijven als

```
(defmethod mapc-musical-object-onset ((object compound-musical-object)

onset operation &rest args)
```

```
"Apply the operation to this object, and to each sub-..-sub-object and their onsets"

(apply operation object onset args)

(loop for element in (elements object)

for element-onset in (onsets object onset)

do (mapc-musical-object-onset element element-onset operation args)))
```

This recursion scheme is the one used implicit by the `draw- musical- object` function. Although distributing the management of the flow of control through the object oriented code is quite natural, each method calling the code to operate on parts of it, in a subtle way it may give rise to lots of code duplication, when many different operations in fact walk through the data structures in the same way. Making the walk itself into an explict method, like we did in `mapc- musical- object- onset`, in a sense we create not a real program, we only capure the behavior of walking through a musical structure and has still to be supplied with an argument which specifies the 'thing to do' while visiting each part[2]. This yields a nice modularity, and allows for dense definitions of complex processes like the draw program that can be written now in a couple of lines.

```
(defmethod draw-musical-object ((object musical-object) onset (window draw-window))

"Draw a graphical piano-roll representation of a musical object"

(mapc-musical-object-onset #'draw-musical-part onset window))


(defmethod draw-musical-part ((object musical-object) onset (window draw-window))

"Draw a graphical representation of the musical object itself"

(draw-boundary object (boundary object onset window) window))


(draw (example)) =>
```

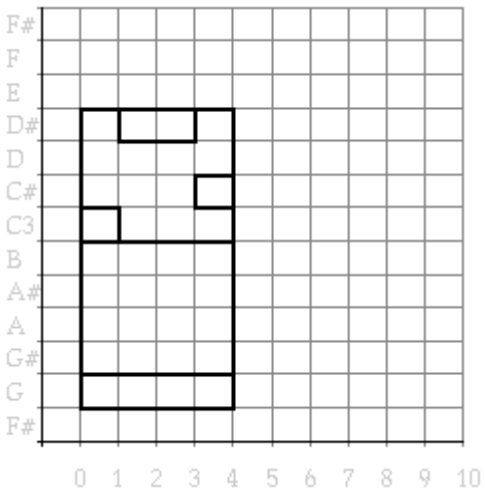

Figure 1. A test, not to appear in book

Of course, a simpler `mapc- musical- object`, which does not calculate onset times can be based in an analogue manner on `mapc- element`.

```
(defmethod mapc-musical-object ((object compound-musical-object) operation &rest args)
"Apply the operation to this object and each sub-..-sub-object"
(apply operation object args)
(apply #'mapc-element-onset object #'mapc-musical-object operation args))

(defmethod mapc-musical-object ((object basic-musical-object) operation &rest args)
"Apply the operation to this object"
(apply operation object args))
```

[to do: This recursor can be trimmed for use by operations that need only be applied to specific objects by adding a predicate. E.g. the transpose (combine-and (wrap #'note?) operation) of hele ding weglaten ?]

And again from iterator to recursor

The final lift of "horizontal" iteration code, to a full recursor definition is needed for `map- elements- onsets`. The difficulty is here that, in contracts to `mapc- elements- onsets` which returns no useful result (it is onnly called for side-effect) and `find- elements- onsets` which returned only one result, this new recursor needs to combine the results at each level of the structure with the results of lower levels and return that to the caller.

```
(defmethod map-musical-object-onsets ((object compound-musical-object)
onset combination operation &rest args)
"Combine the results of the operation applied to each element and its onset"
(apply combination
object
onset
(apply operation object onset args)
(apply #'map-elements-onsets
object
```

```
onset
#'map-musical-object-onsets
combination
operation
args)
args))

(defmethod map-musical-object-onsets ((object basic-musical-object)
onset combination operation &rest args)
"Apply the operation tho the object and its onset"
(apply operation object onset args))
```

If not understood, uitschrijven functional style: can substitueren body van map-elements-onsets recursive call for each element again if not understood substitute body of map-musical-object-onsets

```
(defmethod map-musical-object-onsets ((object compound-musical-object)
onset combination operation &rest args)
"Combine the results of the operation applied to each element and its onset"
(apply combination
object
onset
(apply operation object onset args)
(loop for element in (elements object)
for element-onset in (onsets object onset)
collect (apply #'map-musical-object-onsets element
element-onset
combination
operation
args))
args))
```

Our example that uses this recursor will calculate a flat, non-hierarchical, representation of musical objects, a kind of note list (as is often used in composition systems).

event lists

[to do: tekst aanpassen]

To facilitate translating our musical objects to other representations that use flat agenda-like structures (like standard MIDI files of score files for Csound or Cmusic), we will write a function that can distill an event list, a list that contains pairs of start times and objects in some format. The `event- list` method collects all the calls `object- note- list` on musical objects to obtain the event list which represents by the object itself: a list of the onset-object pair. Compound musical objects only have to append the note list of their components to yield the event list for that object itself.

```
(defmethod note-list ((object musical-object) onset)

"Return a list of onset-event pairs of all parts of a musical object"

(map-musical-object-onsets object onset #'combine-event-lists #'object-note-list))


(defmethod combine-event-lists ((object compound-musical-object) onset self parts)

"Append the event lists of the parts of an object"

(apply #'append (cons self parts)))


(defmethod object-note-list ((object musical-object) onset)

"Return a basis event list consisting of one onet-part pair"

nil)


(defmethod object-note-list ((object note) onset)

"Return a basis event list consisting of one onet-part pair"

(list (list onset object)))


(pprint (note-list (example) 0)) =>

((0 (note :duration 1 :pitch "C3"))

(1 (note :duration 2 :pitch "D#3"))

(3 (note :duration 1 :pitch "C#3"))

(0 (note :duration 4 :pitch "G2")))
```

Simplify to understand, transparency, substitute and sumstitute back

```
(defmethod note-list ((object musical-object) onset)
```

```
"Return a list of onset-event pairs of all parts of a musical object"

(combine-event-lists object

onset

(object-note-list object onset)

(loop for element in (elements object)

for element-onset in (onsets object onset)

collect (note-list element element-onset))))
```

This version is quite specific, it requires a redefinition of `object- note- list` whenever another format is needed. A slightly more general version allows passing each part the method to use for calculating the format of the basic event list. **[to do: parameterize as general method]**

```
(defmethod event-list ((object musical-object) onset object-event-list-method)

"Return a list of onset-event pairs of all parts of a musical object"

(map-musical-object-onsets object

onset

#'combine-event-lists

object-event-list-method))
```

Using two simple methods for `object- note- list` can now, when plugged into the `event- list` function yield an event list which only contains the notes, a result which is needed when the representation that we are converting our musical objects to supports no structure. **[to do: this is the reconstruction of note-list]**

```
(defmethod note-list ((object musical-object) onset)

"Return a list of onset-event pairs of all parts of a musical object"

(event-list object onset #'object-note-list))
```

[to do: voorbeeld met ander formaat hier bv al Cmusic]

[to do: vuistregel: algemene geval is makkelijker]

Though the function succeeds in extracting al notes and their onsets from a structured musical object, the resulting note list does not necessarily appear in time order, a requirement that is often encountered. One solution is to sort the list after it has been made. Note how the `sort` function is used with a proper key to retrieve the onset time as the first of each pair, this will act as the index upon which the list is sorted.

```
(defmethod sorted-note-list ((object musical-object) onset)
"Return a note list: (onset note) pairs, with onsets in ascending order"
(sort-event-list (note-list object onset)))

(defun sort-event-list (event-list)
"Sort a list of onset-object pairs in order of increasing onsets"
(sort event-list #'< :key #'first))

(pprint (sorted-note-list (example) 0)) =>
((0 (note :duration 1 :pitch "C3"))
(0 (note :duration 4 :pitch "G2"))
(1 (note :duration 2 :pitch "D#3"))
(3 (note :duration 1 :pitch "C#3")))
```

However, sorting is an expensive operation, and e.g. the event-lists for the components of a sequential musical object will already all follow each other in time. Appending them together directly is possible, without further sorting. And for parallel objects, assuming that the recursive calls of as-event-list derive event-lists for the components that are sorted themselves, they only have to be merged together, which is a much more efficient processes than sorting.

We need to write the auxiliary functions that combine event lists. Because the predefined `merge` function only can deal with two lists at a time we will need to extend its use to be able to merge a set of sorted lists. The `reduce` function is a handy tool in these situations. It can generalize binary operations (operations of two arguments) to the n-ary case. E.g. `(reduce #'+ '(a b c)) = (+ a (+ b c))`. The `merge` function is first tailored to use for event lists by requiring it to return its result as a list and by supplying it with a key parameter: the function that extracts from each event the aspect on which the sorting has to be done. This is here the time of the event: the first element of the onset-object pair.

```
(defun event-merge (list1 list2)
"Return a sorted event list of all events of both arguments"
(merge 'list list1 list2 #'< :key #'first))
```

[to do: naam event-merge beter kiezen]

```
(defun merge-events (&rest events)
"Merge a list of sorted agenda's into one sorted agenda"
(reduce #'event-merge events))
```

```
(defmethod combine-event-lists ((object parallel) onset self parts)

"Merge a list of sorted agenda's into one sorted agenda"

(apply #'merge-events (cons self parts)))


(defmethod combine-event-lists ((object sequential) onset self parts)

"Concatenate a list of sorted agenda's into one sorted agenda"

(apply #'append (cons self parts)))
```

Now event lists will be guaranteed to be sorted at every level directly, and the top function like `note-list` needs not sort the result anymore - a much more efficient solution.

```
(pprint (note-list (example) 0)) =>

((0 (note :duration 1 :pitch "C3"))

(0 (note :duration 4 :pitch "G2"))

(1 (note :duration 2 :pitch "D#3"))

(3 (note :duration 1 :pitch "C#3")))
```

[to do: remark on optimization as extra specialization methods]

Promoting event lists to musical objects

From here we can go two ways: further upward, trying to incorporate some sort of event list notion into our musical objects (and allow transformations, drawing etc.), and further downward towards simple data structures and score file writers. We will indeed travel both ways, starting with the former. It is claimed often that the object oriented style enables programmers to quickly add small pieces of code that takes care of new situations. We will do so now by introducing a new musical object and integrating it into the code suite we have build up. This will test the flexibility of the classes and methods we defined and form a kind of summary in which the different constructs are used once again.

We want to introduce an object with a more absolute way of specifying the onsets of its elements, similar to an event list. Or alternatively, we want to extend our code, which is completely score-based, a bit more towards performance data. Then we need to define an object that can arbitrarily specify the timing of objects as well - not just a strict parallel or sequential order. It will be a compound musical object, but next to a list of elements, it will hold a list of onset times, one for each element. These onset times will be relative to the start time of the object itself, a nice middle ground between absolutely absolute time specification of flat event lists and a completely relative time specification like in the compound musical objects. We will call his agenda-like structure a `collection`. It can be used in a hierarchical way, nested with another collection or compound musical object and it can contain other collections or compound musical objects. It resembles the event-lists of e.g. SmOKe (Pope, 1992)

closely. The `collection` class can inherit the `elements` slot from the abstract `compound- musical-object` class.

[to do: plaatje hier neerzetten?]

```
(defclass collection (compound-musical-object)

((delta-onsets :accessor delta-onsets :initarg :delta-onsets :initform nil

:documentation "The relative start times of the elements"))

(:documentation "A group of musical objects, starting at specified times"))
```

Now we can define a constructor function for a collection that makes its call easy to read, by alternating onsets and elements.

```
(defun collection (&rest agenda)

"Return a collection of musical objects starting at specified onsets"

(let ((delta-onsets (loop for delta-onset in agenda by #'cddr

collect delta-onset))

(elements (loop for element in (rest agenda) by #'cddr

collect element)))

(make-instance 'collection :delta-onsets delta-onsets :elements elements)))
```

```
(collection 0.3 (note :duration 1 :pitch "C3")

1.7 (note :duration 2 :pitch "B2")

3.1 (note :duration 1 :pitch "C#3"))

->

#<error printing collection #xD80EF9>
```

[to do: which reminds us of printing need to be show invoked because compound]

[to do: definition print-object ophalen]

show method that needs to be defined and, for consistency, needs to reflect the textual representation of the way in which we construct collections.

```
(defmethod show ((object collection))

"Return a list describing a collection of musical objects"
```

```
(cons 'collection

(loop for onset in (delta-onsets object)

for element in (elements object)

collect onset

collect (show element))))
```

We can check the `show` method and define an example for later use. [3]

```
(defun example-collection ()

"Return a simple collection of 5 notes"

(collection 0.0 (note :duration 4 :pitch "G2")

0.3 (note :duration 1 :pitch "C3")

0.7 (note :duration 2 :pitch "D#3")

1.7 (note :duration 2 :pitch "B2")

3.1 (note :duration 1 :pitch "C#3")))


(example-collection)

->

(collection 0.0 (note :duration 4 :pitch "G2")

0.3 (note :duration 1 :pitch "C3")

0.7 (note :duration 2 :pitch "D#3")

1.7 (note :duration 2 :pitch "B2")

3.1 (note :duration 1 :pitch "C#3"))
```

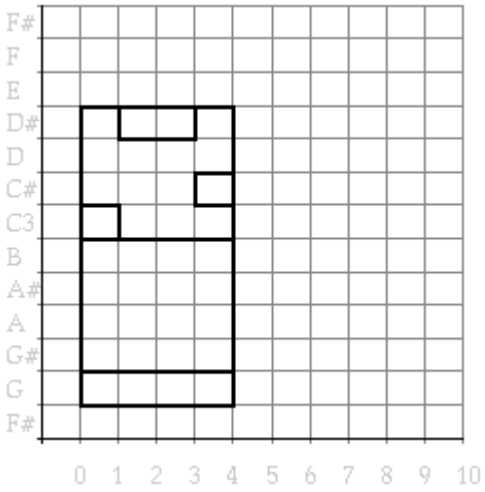

Figure 2. A collection.

This collection is shown in Figure 2, a figure whose drawing has to be postponed till we have defined some more methods to work for collections as well

The accessor function defined on musical objects all have to be checked and possibly defined or re-defined. One of the main ones is the onsets method on which most of the iterators and recursors are based. The onsets of the elements of a collection are calculated by adding their relative onsets to the onset of the collection itself.

```
(defmethod onsets ((object collection) &optional (onset 0))

(loop for element-onset in (delta-onsets object)

collect (+ element-onset onset)))
```

Collections are different than othe compound musical objects in that it makes sense to talk about changing the onset time of their elements. Setf methods can be made to have more arguments, and it is natural to design updates to the onsets of the elements of a collection as having an optional argument that represents the onset time of the object itself.

```
(defmethod (setf onsets) (onsets (object collection) &optional (onset 0))

"Set the onset times of a collection, relative to the onset of the object"

(setf (delta-onsets object)

(loop for element-onset in onsets collect (- element-onset onset))))
```

[to do: &optional ??]

[to do: knippen]

Since onsets and elements are the only access methods needed by the map-elements-onsets method for compound musical objects, functions can be directly based on it

The duration of a collection is calculated by returning the offset time of the component that ends last. A type predicate and the equality check are easily defined as well.

```
(defmethod duration ((object collection))
```

```
"Return the duration of a collection"

(apply #'max (map-elements-onsets object 0 #'offset)))


(defmethod collection? ((object t))

"Return t if the object is an instantiation of class collection, nil otherwise"

(typep object 'collection))


(defmethod same? and ((object1 collection)(object2 collection))

"Are two two parallel musical objects equal?"

(and (equal (delta-onsets object1)(delta-onsets object2))

(same-set (elements object 1)(elements object2) :test #'same?)))


(duration (example-collection))

->

4.1
```

Drawing now works

To make `retrograde` work for collections as well, a method definition needs to be added. It calculates the time left from the end of each component till the end of the whole collection. These times form the new onsets of the components of a retrograde collection. The result of this transformation applied to a collection is shown in Figure 3.

```
(defmethod retrograde ((object collection))

"Reverse all components of the object in time"

(setf (onsets object)

(loop with duration = (duration object)

for element in (elements object)

for element-onset in (delta-onsets object)

collect (- duration

(offset element element-onset))))

object)
```

```
(draw (original-and-transform (example-collection)

(pause :duration 1)

#'retrograde)) =>
```

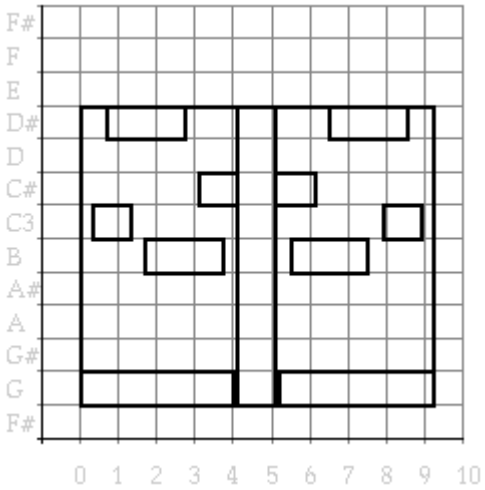

Figure 3. The collection and its retrograde.

Because show works well, so will the `copy- object` and thus the `repeat`, `canon`, and `original- and-transform` functions. Most transformations rely on one of the mappers, and the definition of the mappers for compound musical objects rely only on the availability of an `onsets` method to calculate proper onset times for the elements. Thus the definitions of the mappers will turn out to work just fine for collections. [4] And `transpose` and `mirror` etc. will work well for collections automatically, because they only need a well defined iterator, as will `draw`. One other function that has to be written is the `combine- event-lists` method for collections. Collection behaves like parallel structures in this respect: when event lists of their components are to be combined, they have to be mergerd in the proper order.

```
(defmethod combine-event-lists ((object collection) onset self parts)

"Merge a list of sorted agenda's into one sorted agenda"

(apply #'merge-events (cons self parts)))
```

All our compound musical objects can be converted to a collection. In this sense a collection is a more general data structure. However, a collection tells less about the time order of the components, it is less specific. [5]

An around method can prevent a call of the primary method

Consider a protection that would be desirable for modifications of the pitches of musical objects. By mistake a transposition or mirror operation can easily create pitches outside a permitted range. E.g. for

MIDI the pitches should be between 0 and 127. Lets try what happens when we create note that is too low.

```
(pitch (transpose (note :pitch "C1") -40))

->

-4
```

Of course we would not be doing CLOS if we were not to be looking for way in which we can program a prper protection without changing the code of `transpose`. A first approach could be to cause an error in a method before the harm is done.

```
(defmethod transpose :before ((object note) interval)

"Break when the transposition would move the pitch outside the MIDI range"

(unless (<= 0 (+ (pitch object) interval) 127)

(error "~A cannot be transposed by ~A semitones" object interval)))
```

```
(pitch (transpose (note :pitch "C1") -40))

=>

> Error:

> (note :duration 1 :pitch "C1") cannot be transposed by -40 semitones

> While executing: #<standard-method transpose :before (note t)>
```

This is not a bad approach, and in many cases it is appropriate. However, would we like to ignore the update and proceed with the rest of the calculations, this method would not work. For these situations CLOS supports a particular kind of method which can control the invocation of other methods: the around method. When such a method (with the qualifier `:around`) is among the applicable methods, that method will be called instead of any before, after or primary one. And it can decide to do all calculation itself and return a result. However, the around method can also actively ask for the primary method (plus possible before and after methods) to run and obtain the result of the primary method. It can do so by evaluating the form `(call- next- method)`. In this sense the method combination is not declarative anymore: calling the primary method is under direct control of the program. When the `(call- next- method)` form is evaluated, the primary method is called with the same arguments as the around method itself. This mechanism can solve our pitch out-of-range protection, the real transposition is only invoked when it is save to do so. Remember that we made transformations always return their object, thus even in this pathological case, whe the transformation refuses to work, we need to adhere to that protocol.

```
(defmethod transpose :around ((object note) interval)

"Don't transpose when this would move the pitch outside the MIDI range"

(if (<= 0 (+ (pitch object) interval) 127)
```

```
(call-next-method)

object))


(transpose (note :pitch "C1") -40)

->

(note :duration 1 :pitch "C1")
```

Recognizing that the `mirror` transform needs a similar protection, we really want to shift the guard further downward to prevent any illegal pitch update. Since all modifications of the pitch slot of a note are made using a `setf` writer method which was generated automatically by the system, and we already know these *setf methods* are defined, it is no big step to write a before setf method to properly disallow all illegal pitch updates. Remeber to always let `setf` return its value.

```
(defmethod (setf pitch) :around (value (object note))

"Call the primary method only with a pitch that is kept within [0,127]"

(if (<= 0 value 127)

(call-next-method)

value))


(mirror (note :pitch "C1") "C5")

->

(note :duration 1 :pitch "C1")
```

In a primary method the next method may be called as well, but please don't

The `call- next- method` form can be used in any primary method too (not only in around methods), calling the most specific method shadowed by this one. This fact is considered an invitation to programmers to write ugly code for which actual method combination may become a hard puzzle of control-flow, especially since the decision whether a call to `call- next- method` is done depends on the actual computation, not just on the program text. Therefore it is better to avoid this use of `call- next- method`.

An around method can change the arguments of its primary method

There is still another way to protect pitch updates outside of the allowed range: by first clipping the new value to be inside that range. Thus we would like to call the update method but now with different arguments. Around methods can do that by calling `call- next- method` with a new set of arguments. This is a fine way to wrap a protection around an assignment.

```
(defmethod (setf pitch) :around (value (object note))

"Call the primary method with a pitch that is kept within [0,127]"

(call-next-method (clip 0 value 127) object))
```

[to do: import this function]

```
(defun clip (min value max)

"Return the value, clipped within [min,max]"

(cond ((< value min) min)

((> value max) max)

(t value)))


(pitch (mirror (note :pitch "C1") "C5"))

->

127
```

As a last extension we can allow the pitch to be set to a pitch name directly, a translation that we had already implemented in the initial note construction.

```
(defmethod (setf pitch) :around (value (object note))

"Call the primary method with a pitch number"

(let* ((number (if (stringp value)(pitch-name-to-midi-number value) value))

(safe-number (clip 0 number 127)))

(call-next-method safe-number object)))


(let ((object (note)))

(setf (pitch object) "C#5")

object)

->

(note :duration 1 :pitch "C#5")
```

An around method can modify the results returned by the primary method

Now we can try a rough solution to the problem that the boundary boxes in our piano-roll drawings do not nest properly. It is obtained by enlarging the boundary enclosing each object a bit before it is returned.

This is an example of the use of an around method that amends the return value of the primary method. is not at all a trivial task to define an algorithm that draws the hierarchical structure, the boxes around compound musical objects, in such a way that edges do not overlap. That is because the diagramatic time-pitch plot of the piano-roll notation does not leave space for depicting the logical hierarchical structure. But a quite nice solution can be found if we agree on stealing a bit of space from some note boxes (and if we disregard the possibility of parallel grouping with largely overlapping pitch ranges). **[to do: dit is niet waar hier, no stealing]**The result of this kind of drawing is shown in Figure ? that can be compared to the original Figure 7 which had no nesting. [6]

```
(defclass nested-mixin (draw-window-mixin)

()

(:documentation "A mixin to draw the structure in a nested fashion"))


(defmethod boundary :around ((object compound-musical-object) onset (window nested-
mixin))

"Draw the boundary as side effect of its calculation"

(declare (ignore onset))

(enlarge-rectangle (call-next-method)

(nested-margin object window)))


(defmethod nested-margin ((object musical-object) (window nested-mixin))

"The margins of a nested compound boundary box"

(let ((horizontal .15)(vertical .2))

(list (- horizontal) vertical horizontal (- vertical))))


(defun enlarge-rectangle (box margins)

(mapcar #'+ box margins))


(defclass draw-window (mouse-select-mixin

nested-mixin

structured-mixin

visible-selection-mixin

piano-roll-contents-draw-window)

()

(:documentation "A new pianoroll window class"))
```

```
(draw (original-and-transform (example)

(pause :duration 1)

#'mirror 60)) =>
```

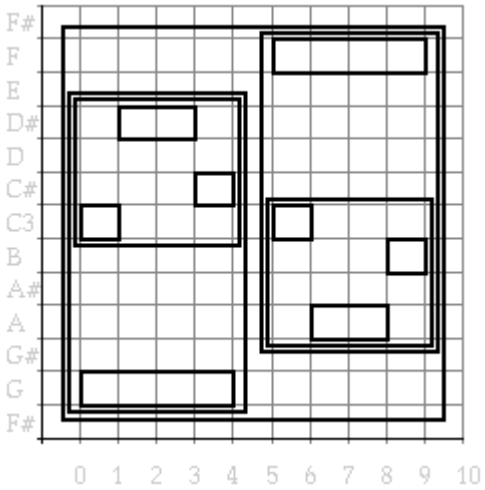

Figure 4. Nested drawing

This complicated program change for the nested drawing of musical objects thus boils down to the modification of the calculation of the boundary of a box by adding one around method. Apart from the clever design of the CLOS language that makes this possible, it illustrates how the use of fine grained modularity in writing programs, pays itself back in ease of program modification. The modification would have been much harder and more error prone had we not introduced an auxilary boundary method in the body of the original draw object methods. Students of object oriented programming should train the sensitivity to spot the places in a program where an extra hook, in the form of the call to an auxilary function, could become indispensible in later extentions or modifications. Reusing and rewriting pieces of code a lot, and reflecting on the cases in which reusing was not possible due to design flaws makes good practice for increasing that sensitivity. Reusable code is too often a commercial slogan instead of an approriate description of object oriented programs - that should change.

[to do: wat kleiner maken noten aan de hand van diepte, met shrink-boundary ?]

Naming musical objects

In searching through musical objects, the compound objects can only be distinguished by onset time and duration and the pich range they cover. In modeling musical knowledge using our classes, it would be nice to be able to distinguish, say a sequential structure representing a bar, from one representing a voice of a piece. Instead of incorporating much more musical knowledge into our class hierarchy[7] we will introduce names for musical objects. These names can be the link to knowledge that will be kept outside the system itself. The name slot can be added to the existing class definition for compound musical objects.

```
(defclass named-musical-object-mixin ()
((name :accessor name :initarg :name :initform nil))
(:documentation "A named object"))

(defclass compound-musical-object (musical-object named-musical-object-mixin)
((elements :accessor elements
:initarg :elements
:documentation "The components"))
(:documentation "A named group of musical objects ordered in time"))
```

To allow the name slot to be filled when we create a compound object, we define some new creators that are similar to the old ones, but have a name as first argument and pass it to make-instance.

```
(defun S (name &rest elements)
"Return a named sequential musical object"
(make-instance 'sequential :name name :elements elements))

(defun P (name &rest elements)
"Return a named parallel musical object"
(make-instance 'parallel :name name :elements elements))

(defun C (name &rest agenda)
"Return a named collection"
(let ((onsets (loop for delta-onset in agenda by #'cddr collect delta-onset))
(elements (loop for element in (rest agenda) by #'cddr collect element)))
(make-instance 'collection :name name :delta-onsets delta-onsets :elements elements)))
```

This allows an example to be defined with some names.

```
(defun named-example ()
"Returns a simple musical object with named structure"
(P "Example" (S "Melody"
```

```
(note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

(note :duration 4 :pitch "G2")))
```

The `show` method can be patched with an around method in case of named objects if so, it builds a list in the format according to the new creators, otherwise it uses the old ones. An around method ain is the way to get hold of the result returned by our old primary `show` method.

```
(defmethod show :around ((object named-musical-object-mixin))

"Add the constructor and possibly the name to the showing of the elements"

(if (name object)

(list* (named-constructor object) (name object) (rest (call-next-method)))

(call-next-method)))
```

```
(defmethod named-constructor ((object sequential)) 's)

(defmethod named-constructor ((object parallel)) 'p)

(defmethod named-constructor ((object collection)) 'c)
```

[to do: of al de functie constructor-name schrijven, goed voorbeeld voor method and combination]

Because we have already bound show into the system printing, the definitions will be used directly for any output of named musical objects.

```
(named-example)

->

(P "Example" (S "Melody"

(note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

(note :duration 4 :pitch "G2"))
```

For printing the names in the piano roll notation we need to reserve some space, extending the boundary a bit upward, whenever a compound musical object has a name. An extra before method for `draw-boundary` can print the name at the appropriate place. The new around method is ‘wrapped around’ other possible around methods, in this case the around method that is responsible for proper nesting of the

boxes. This makes the new behaviour properly combine with the old, and even the mouse sensitivity and visible selections are made in a consistent way, reflecting the new largere boundaries

```
(defclass named-mixin (draw-window-mixin)

()

(:documentation "A mixin to print name tags of objects"))
```

```
(defmethod boundary :around ((object compound-musical-object) onset (window named-
mixin))

"Draw the boundary as side effect of its calculation"

(declare (ignore onset))

(enlarge-rectangle (call-next-method)

(named-margin object window)))
```

```
(defmethod named-margin ((object compound-musical-object) (window named-mixin))

"Leave space in a boundary box for the name"

(if (name object)

(list 0 .7 0 0)

'(0 0 0 0)))
```

[to do: this should be like (window-font-height window)]

```
(defmethod draw-boundary :after ((object compound-musical-object) boundary (window
named-mixin))

"Draw a label in the box when a compound musical object is named"

(when (name object)

(draw-label window boundary (name object))))
```

[to do: de enlarge aanroep in een method van de algemen window class dan kan deze ook zonder nested]

```
(defclass draw-window (named-mixin

mouse-select-mixin

nested-mixin

structured-mixin

visible-selection-mixin

piano-roll-contents-draw-window)
```

```
()

(:documentation "The window class named pianoroll"))


(draw (named-example)) =>
```

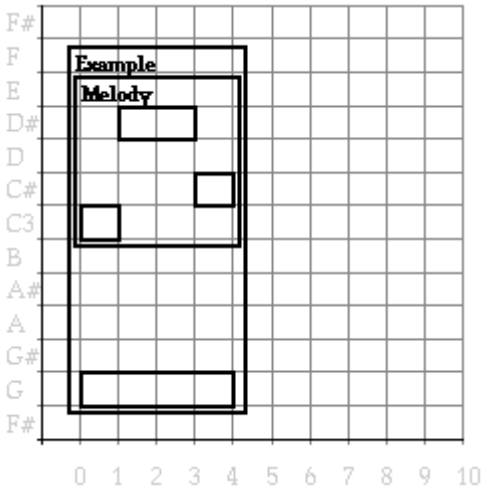

Figure 5. Named musical objects.

On top of the search function we can now define the search for an object with a specific name, e.g. to subject it to a transformation.

```
(defun named (name)

#'(lambda(object onset)

(when (and (name object)

(string-equal name (name object)))

object)))


(draw (named-example))

(front-select (named "Melody")) =>
```

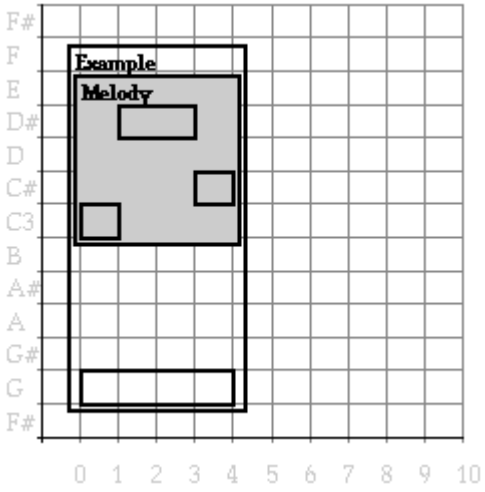

Figure 6. Selection by name

opmerking over before after, draw boundary + selection all draw acties volgorde mixin volgorde geen protocol

initialize instance

In CLOS all the work that is done during creation and initialization of objects is carried out by a few methods (like make-instance) that can be augmented or modified by the programmer. As an example, we will implement a check on the sanity of the compound structures after they is created. The routine that is called by the system after creation of an instance is initialize-instance. It is called with as arguments the object to be initialized and a list of keyword value pairs from (among others) the make-instance call. We will define an after method for our check, remembering to use a compatible lambda list.

```
(defmethod initialize-instance :after ((object compound-musical-object) &rest
arguments)

"Check that each element of a parallel musical object has the same duration"

(declare (ignore arguments))

(loop for element in (elements object)

unless (typep element 'musical-object)

do (error "musical object may not contain a ~A: ~A" (type-of element) element)))
```

This method will be run by the system after creating an instance of the class parallel. When the test on the equalitity of the duration of the components fails, the execution will be halted with an error message. Note how no attempt is made to print the whol musical object: our show method would choke on the element just detected.

```
(parallel (note) 1.3 (pause :duration 3) nil)

=>

> Error: musical object may not contain a double-float: 1.3
```

[to do: dubbele text ?, doorschrijven]

customizing initialization

After CLOS has instantiated a class, but before returning the object as the result of `make- instance`, the object is subjected to the `initialize- instance` method. This method is given also all the initialization arguments presented to `make- instance`. This is the perfect place to add some of our own initialization behavior. For example, we have prevented pitch updates outside the reasonable range, but we haven't prevented the creation of a note with such a pitch. Because our setf around method can deal with the details of clipping the pitch, we only have to give it a chance to do so.

```
(defmethod initialize-instance :after ((object note) &rest args &key pitch &allow-
other-keys)

"Allow the pitch asignment protection to do its work at creation time as well"

(setf (pitch object) pitch))
```

The above will take care of the translation of a pitch name to a midi value as well. The other type of initization preprocessing, the traetment of an initial frequency specification, that we supported in a rather ugly way in de note constructor function, can no also beget a more elegant form. This makes our tinkering with the constructor function itself superfluous: all initialization is handled by the `initialize- instance` after method.

```
(defmethod initialize-instance :after ((object note) &rest args

&key pitch

frequency

&allow-other-keys)

"Allow the pitch asignment protection to do its work at creation time as well"

(when pitch (setf (pitch object) pitch))

(when frequency (setf (frequency object) frequency)))


(defun note (&rest attributes)

"Return a note object with specified attributes"

(apply #'make-instance 'note attributes))
```

```
(note :frequency 440)

->

(note :duration 1 :pitch "A3")
```

Setf methods

Just like the duration of a basic musical object can be updated, it may be necessary to change the duration of compound musical objects, compressing or expanding all their elements. If we were to write a procedure that establishes that, called say `set- duration`, the programmer has to remember that durations of basic musical objects are changed using `setf` on the duration slot, and that the `set-duration` procedure has to be used to change the durations of compound musical objects. It is again best to make a uniform interface for updating, by writing a setf method for duration, specialized for compound musical objects. This method needs to calculate a stretch factor (a ratio between desired and actual duration) and elongates (or shrinks) the duration of all its elements with that factor. The use of mutual recursion between the `(setf duration)` method and the `stretch` transformation comes quite natural here.

```
(defmethod (setf duration) (value (object compound-musical-object))

"Set the duration of a compound musical object by stretching its components"

(let ((factor (/ value (duration object))))

(mapc-elements object #'(lambda(object)(stretch object factor)))

value))
```

```
(defmethod stretch ((object musical-object) factor)

"Multiply the duration of a musical object by a factor"

(setf (duration object) (* (duration object) factor))

object)
```

Remember that the order in the argument list of the `setf` method is first the value and then the object and that a `setf` method should always return its value. The `stretch` transformation that changes the duration of a musical fragment is not just an auxiliary function for the `(setf duration)` method, it is useful in itself. In Figure 7 it is applied to the example to yield a fast version.

```
(draw (original-and-transform

(example)

(pause :duration 1)

#'stretch .5)) =>
```

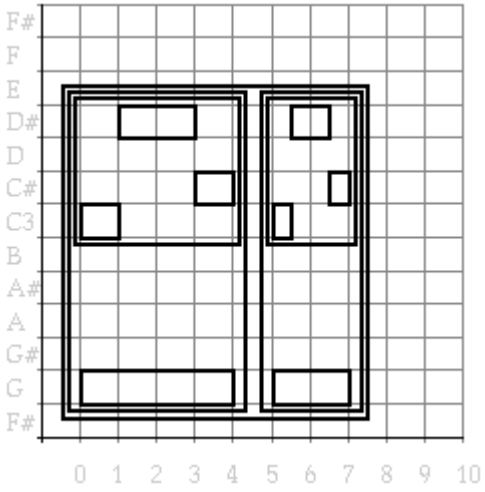

Figure 7. The example and a shortened version.

The definition of a modification macro for multiplications makes the code look more like the other transformations (e.g. `mirror`)

```
(define-modify-macro multf (args) * "Multiply a generalized variable")


(defmethod stretch ((object musical-object) factor)

"Multiply the duration of a musical object by a factor"

(multf (duration object) factor)

object)
```

The `setf` method for duration defined for compound musical objects works a not well for collections: it shrinks or stretches the duration of all its components, but after that, the onsets of the components, which are stored as data in a collection, need to be adapted as well.

```
(defmethod (setf duration) (value (object compound-musical-object))

"Set the duration of a compound musical object by stretching its components"

(let ((factor (/ value (duration object))))

(mapc-elements object #'(lambda(object)(stretch object factor)))

(setf (delta-onsets object)

(loop for delta-onset in (delta-onsets object)

collect (* delta-onset factor)))
```

```
value))


(draw (original-and-transform (example-collection)

(pause :duration 1)

#'stretch .5)) =>
```

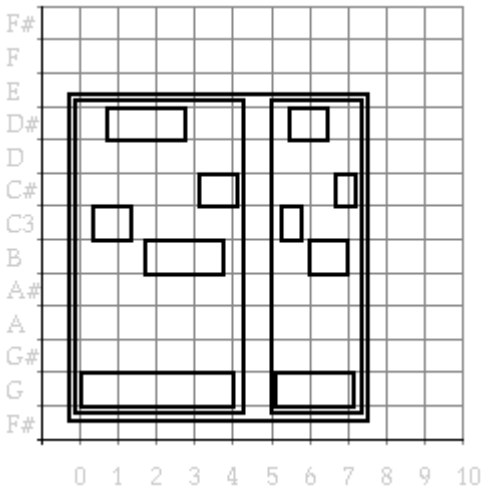

Figure ?. The collection and a shortened version.

Arguments to user commands

The stretch operations seems like a useful addition to our menu of transfomations that can be applied to musical objects in windows. However, it is difficult to find a fixed stretch factor that will satisfy all. We will use one of the turnkey dialogs to have the user enter a stretch factor and use that in the transformation.In case of user input, it is wise to put in some extra checks on the validity of the input.

```
(defmethod stretch-selection ((window selection-mixin)

&optional (factor (get-factor)))

"Stretch the duration of the musical object selected in the window by a factor"

(selection-operation window #'stretch factor))


(defun get-factor ()

"Ask user to type a factor and return it as positive number"

(loop as factor = (read-from-string (get-string-from-user "Factor (> 0)") nil nil)

when (and (numberp factor)(> factor 0))

do (return factor)))
```

For consistency we will also change the mirror operation and add a general transpose operation along the same lines.

```
(defmethod transpose-selection ((window draw-window)
&optional (interval (get-interval)))
"Transpose the musical object selected in the window by an interval"
(selection-operation window #'transpose interval))

(defun get-interval ()
"Ask user to type an interval in semitones and return it as number"
(loop as interval =
(read-from-string (get-string-from-user "Interval (in semitones)")
nil
nil)
when (numberp interval)
do (return interval)))

(defmethod mirror-selection ((window draw-window))
"Reflect the musical object selected in the window around a center"
(selection-operation window #'mirror (get-pitch)))

(defun get-pitch ()
"Ask user to type an pitch name"
(get-string-from-user "Pitch name"))
```

Now the new commands can be added to the menu and tried.

```
(add-music-command "Transpose" #'transpose-selection-up #\t)
(add-music-command "Mirror" #'mirror-selection #\m)
(add-music-command "Stretch" #'stretch-selection #\S)

(draw (example))
```

```
(front-mouse-click .5 60 nil 1) =>
```

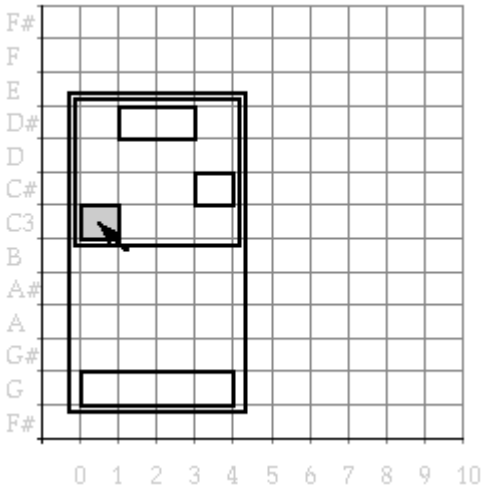

Figure ?. A. Test of the stretch operation

```
(front-draw-window) =>
```

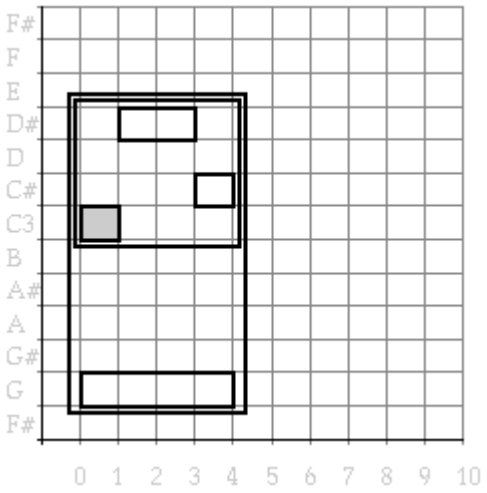

Figure ?. B. Test of the stretch operation

```
(front-draw-window) =>
```

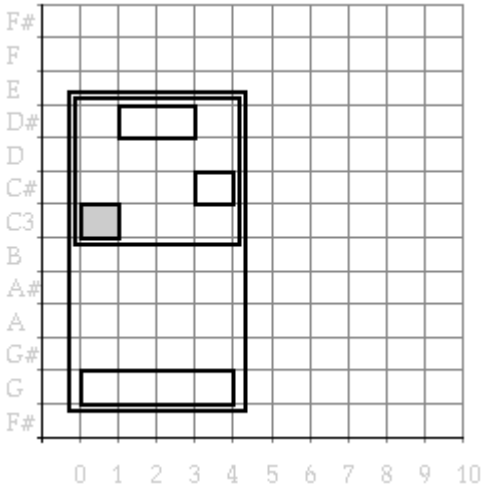

Figure ?. C. Test of the stretch operation

```
(front-draw-window) =>
```

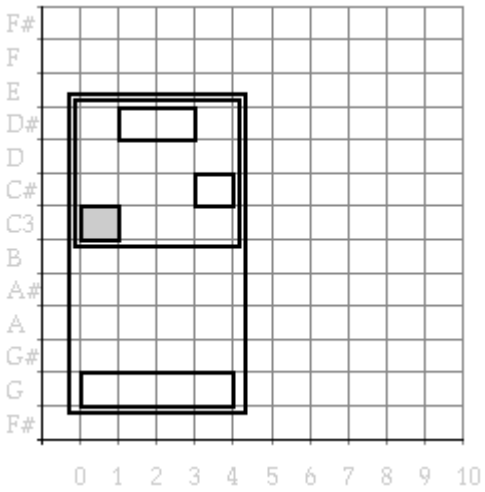

Figure ?. D. Test of the stretch operation

```
(do-front-window #'stretch-selection 2.5)

(front-draw-window) =>
```

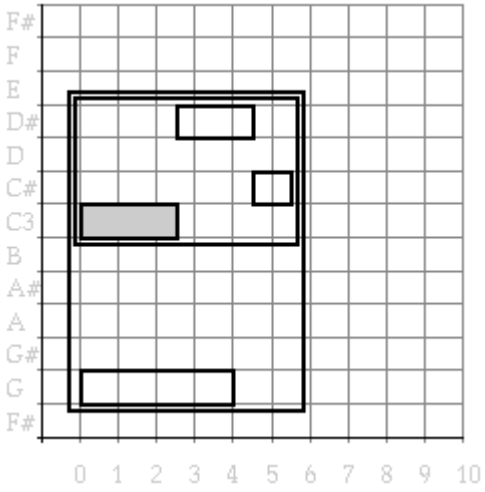

Figure ?. E. Test of the stretch operation

Integer note maken en naar boven schuiven, na mixin, na after around, na after init, voor change window class

All notes would work as well with fraction apitch numbers, niet gelijk zwevende stemmingen etc transpose, intonatie. Maar MIDI kan niet aan, dus moet afronden. later zal met allocatie op kanalen wat verlichten. Support as mixin for notes. To guaranty a proper creation of a midi note, we write an after method for the initialisation

```
(defclass MIDI-note-mixin ()

()

(:documentation "A note mixin which will guarantee an integer pitch number"))


(defmethod initialize-instance :after ((object MIDI-note-mixin) &rest args)

(setf (pitch object) (round (pitch object))))
```

A concrete class and a constructor is easily defined, after which we can check the creation of

```
(defclass MIDI-note (MIDI-note-mixin note)

()

(:documentation "A note which will guarantee an integer pitch number"))


(defun MIDI-note (&rest args)
```

```
(apply #'make-instance 'MIDI-note args))


(defun MIDI-note? (object)

(typep object 'MIDI-note))


(pitch (MIDI-note :pitch 60.3))

->

60
```

Again we can write an setf around method to correctly implement any later updating of the pitch too.

```
(defmethod (setf pitch) :around (value (object MIDI-note-mixin))

(call-next-method (round value) object))


(let ((note (MIDI-note)))

(setf (pitch note) 60.3)

note)

->

(note :duration 1 :pitch "C3")
```

Now, one of the wonders of CLOS is the possibility to change the class of an existing object. Below we update the pitch and print the note before and after its class has changed to`MIDI- note`.

```
(let ((object (note)))

(setf (pitch object) 61.9)

(print (pitch object))

(change-class object 'MIDI-note)

(setf (pitch object) 61.9)

(print (pitch object)))

=>

61.9

62
```

To prevent inconsistency we of course need the the proper hooks to have our own programs deal with such complex cases. After CLOS changed the class of an object, it call the shared-initialize method reinitialize ?? This dummy method can of course be decorated with an after method to repair the situation in which a note with a fractional pitch is changed to class MIDI-note. We rely on our setf method to do the rounding. or reinitialize instance for new class ??

```
(defmethod update-instance-for-different-class :after

((old note)(new MIDI-note) &rest initargs)

(setf (pitch new)(round (pitch old))))


(let ((object (note :pitch 60.3)))

(print (pitch object))

(change-class object 'MIDI-note)

(print (pitch object)))

=>

60.3

60
```

```
(defun intonation-example ()

(parallel (sequential (note :duration 1 :pitch 60.2)

(note :duration 2 :pitch 62.9)

(note :duration 1 :pitch 61.3))

(note :duration 4 :pitch 55.7)))


(draw (intonation-example)) =>
```

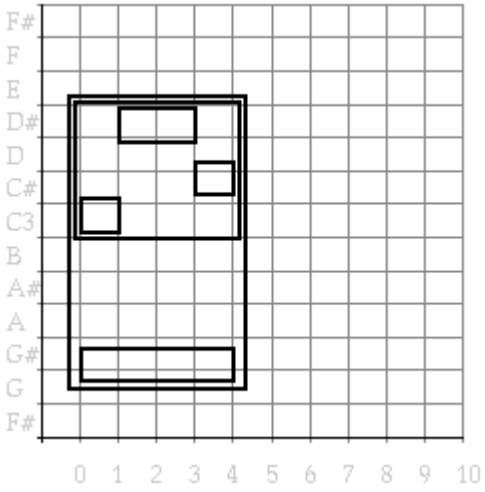

Figure ?. Use of fractional pitches

Changing notes to midi notes and vise versa by the suer can be supported by a menu command workining on the notes in a selection.

```
(defmethod change-note-class ((object compound-musical-object) class)

(mapc-elements object #'change-note-class class))


(defmethod change-note-class ((object note) class)

(change-class object class))


(defmethod change-note-class ((object pause) class)

)
```

(add-music-command “Change to MIDI” #’midi-selection #) (add-music-command “Free from MIDI” #’unmidi-selection #)

```
(defmethod midi-selection ((window draw-window))

(when (selection window)

(change-note-class (selection window) 'MIDI-note)))


(defmethod unmidi-selection ((window draw-window))

(when (selection window)
```

```
(change-note-class (selection window) 'note)))
```

eventueel ook anders eruit zien op midi-awareness-mixin

```
(defclass midi-awareness-mixin (draw-window-mixin)

()

(:documentation "A mixin to show distinction between MIDI and free notes"))


(defmethod draw-boundary ((object note) boundary (window midi-awareness-mixin))

(cl-user::draw-round-rectangle window boundary 15 2 ))


(defmethod draw-boundary ((object MIDI-note-mixin) boundary (window midi-awareness-
mixin))

(draw-rectangle window boundary))


(defclass draw-window (midi-awareness-mixin

named-mixin

mouse-select-mixin

nested-mixin

structured-mixin

visible-selection-mixin

piano-roll-contents-draw-window)

()

(:documentation "The window class named pianoroll"))
```

[to do: als selected dan ook afgerond grijs]

[to do: met mooie muis click windows]

```
(draw (intonation-example))

(front-mouse-click 3.6 61.2 nil 1) =>
```

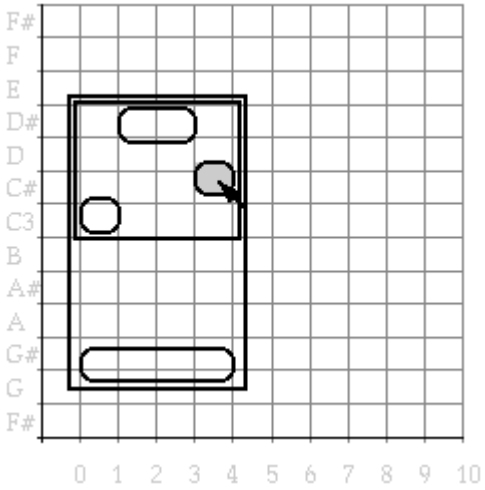

Figure ?. A. changing the class of the notes in the selection

```
(front-draw-window) =>
```

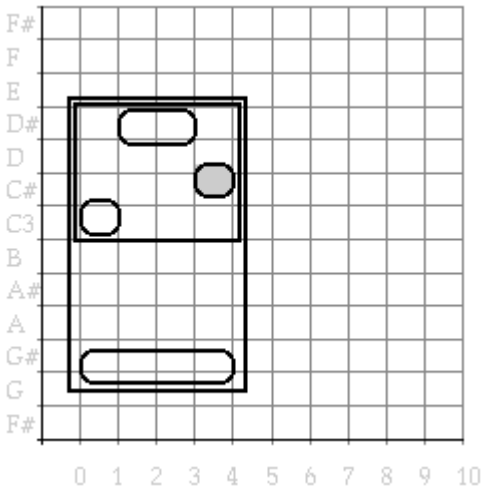

Figure ?. B. changing the class of the notes in the selection

```
(front-selection-operation #'change-note-class 'MIDI-note) =>
```

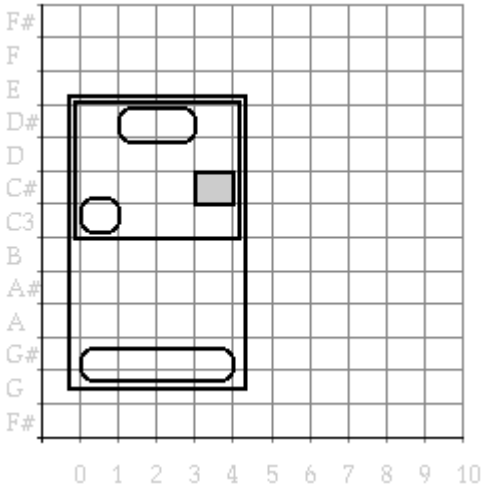

Figure ?. C. changing the class of the notes in the selection

Note how for midi notes on a midi-awareness-mixin the draw-boundary method shadows the less specific method for any note which in turn shadows the method for notes on ... window (because the midi-awareness-mixin appear before the ... window in the list of superclasses of the draw window.

Meta Object Protocol: introspection allows objects to know their class

[to do: in dit stukje een Marx grapje]

Recall the two show methods for compound musical objects:

```
(defmethod show ((object sequential))
"Return a list describing a sequential musical object with its components"
(cons 'sequential
(mapcar #'show (elements object))))

(defmethod show ((object parallel))
"Return a list describing a parallel musical object with its components"
(cons 'parallel
(mapcar #'show (elements object))))
```

They are identical, apart from the name of the class that is put in the front of the list. If we had a way to get hold of the name of the class of an object, one method would suffice. In pure CLOS that is not possible: the so-called programmer interface does not allow explicit access to constructs like classes.

However, part of the beauty of CLOS lies in the so called meta-object protocol. At this level all CLOS constructs (classes, methods etc.) are again implemented as CLOS objects, and thus accessible for the programmer. A simple use of this layer, the so called introspective part, only retrieves innocent objects, like names of classes of objects for use by the programmer. These facilities will be explored a bit first.

```
(defmethod show ((object compound-musical-object))

"Return a list describing a compound musical object with its components"

(cons (class-name (class-of object))

(mapcar #'show (elements object))))
```

Another introspective use of the meta object protocol is to retrieve an inheritance tree of a class. So we are truely having a program inspect the structure of our programs. The `class- subclasses` is by the way also the function used to generate he inheritance diagrams in this chapter automatically.

```
(defun class-subclasses (class)

"Return the tree of classes inheriting from this one"

(cons (class-name class)

(loop for sub-class in (class-direct-subclasses class)

collect (class-subclasses sub-class))))
```

The function must be called on a true class meta object, not a class name. The `find- class` function finds such a class with a given name if it is defined.

```
(pprint (class-subclasses (find-class 'musical-object)))

=>

(musical-object

(compound-musical-object

(collection)

(parallel)

(sequential))

(basic-musical-object

(pause)

(note)))
```

Nice to inspect our growing window class hierarchy, see shaded not used in draw window because

shading used for selection, how both mouse-select and visible-select supply selection mixin behavior to draw window

```
(pprint (class-subclasses (find-class 'draw-window-mixin)))

=>

(draw-window-mixin

(named-mixin (draw-window))

(nested-mixin (draw-window))

(combine-boundary-mixin

(structured-mixin (draw-window)))

(shaded-mixin)

(selection-mixin

(mouse-select-mixin (draw-window))

(visible-selection-mixin (draw-window))))
```

when a flat list is needed, no tree, easy to adapt the function

```
(defun all-class-subclasses (class)

"Return the tree of classes inheriting from this one"

(cons (class-name class)

(remove-duplicates (loop for sub-class in (class-direct-subclasses class)

append (all-class-subclasses sub-class)))))
```

(all-class-subclasses (find-class ‘draw-window-mixin)) (draw-window-mixin named-mixin nested-mixin combine-boundary-mixin structured-mixin shaded-mixin selection-mixin mouse-select-mixin visible-selection-mixin draw-window)

Next to introspection, the meta object protocal can be put to extremely powerful use in extending the CLOS language itself, as will be the topic of the next section.

Meta Object Protocol: mixing class cocktails

Somehow it feels not completely right to have to redefine our draw window each time we have invented a new mixin for it. The fact that CLOS only supports creating an instance from a predefined class prevents us here from being able to create an object of a list of ingredient classes directly. Because we programmers always try to push the limits of the language we have to work with, let us try some clever but dirty kludges to achieve our class coctails, before the use of the meta object protocol that was

designed for elegant ways of pushing the language boundaries.

The first realisation is that a defclass form is just a lisp list, which we know very well how to construct. And evaluating a class definition created by our program should yield a new class accoring to our wishes. We will use `gensym` to have lisp invent a name for it and return it us. [8]

```
(defun define-cocktail-class (ingredients)

(let ((name (gensym)))

(eval (list 'defclass name ingredients ()))

name))
```

Now we need to instantiate an object directly from a cocktail, passing on any initialisation arguments needed.

```
(defun make-cocktail-instance (ingredients &rest args)

(apply #'make-instance (define-cocktail-class ingredients) args))
```

Because make-instance itself is also a method that can be specialised, we can do a bit better in plain CLOS, by specializing its first class argument to a list. In that way we have supplied a clean interface to the user of class cocktails, even though our implementation is still lousy.

```
(defmethod make-instance ((ingredients list) &rest args)

(apply #'make-instance (define-cocktail-class ingredients) args))
```

```
(draw (named-example) :window-class '(named-mixin

nested-mixin

structured-mixin

piano-roll-contents-draw-window)) =>
```

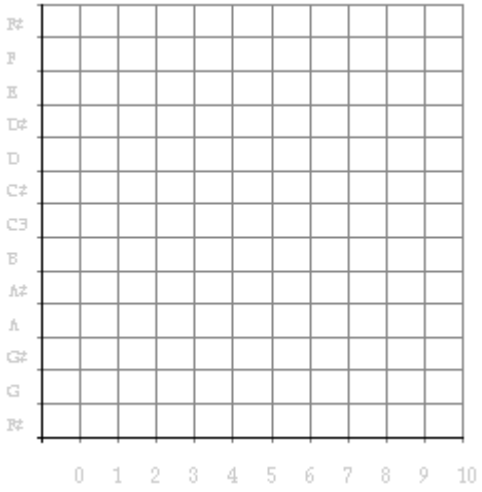

Figure ?. A. Display of the same example on different cocktail windows

Figure ?. B. Display of the same example on different cocktail windows

Now we can proceed to define a CLOS class in a neatly programmed fashion. Below the layer of macro's (like defclass) that are intended to have constant arguments written by the programmer and appearing at the top level in the program, there is a layer of functions which do the real work, and which can of course be called by use directly to construct a class definition under program control. Because classes are, at the meta level, just ordinary CLOS objects, they can be created by make inststance as well. [9]That can also be implemented easily using the metaobject protocol. The class of a class is called `standard- class`. To create an instance of that meta class we have to supply, next to its name, a list of classes of which inherits. Remember that on this level we are always dealing with class meta objects: class names do not suffice.

```
(defun define-cocktail-class (ingredients)

(make-instance 'standard-class

:name (gensym)

:direct-superclasses (mapcar #'find-class ingredients)

:direct-slots nil))

(defun define-cocktail-class (ingredients)

(let* ((name (intern (string (gensym "COCKTAIL-"))))

(class (make-instance 'standard-class

:name name
```

```
:direct-superclasses (mapcar #'find-class ingredients)

:direct-slots nil)))

(setf (find-class name) class)

name

))
```

To create a window class of our choice, we could now program a menu. Since all our drawing mixins inherit from `draw- window- mixin` we can lok those up and thus even make the menu dynamic: when new draw mixins arise they will automatically appear in the menu. For extra juicyness we display not the name of the class but its documentation.

```
(defun select-direct-subclasses (class)

(mapcar #'first

(select-item-from-list

(loop for class in (class-direct-subclasses (find-class class))

as name = (class-name class)

collect (list name (documentation name 'type)))

:window-title "Ingredients"

:selection-type :disjoint

:table-print-function #'(lambda(item stream)(princ (second item) stream)))))


(defun get-window-cocktail ()

(define-cocktail-class (append

(select-direct-subclasses 'draw-window-mixin)

'(piano-roll-contents-draw-window))))
```

use command to multiple select

```
(draw (example) :window-class (get-window-cocktail))
```

[to do: plaatje van het menu]

The class cocktail mixer cannot only be used to create new window classes, an existing window can also be instructed to change it class. Thus we can alter the appearence of our musical objects on the fly.

```
(defmethod window-change-class ((object draw-window-mixin) &optional (class (get-
window-cocktail)))

(change-class object class))



(add-music-command "Change appearence" #'window-change-class)
```

[to do: refinements, betere gensym, documentation ophalen en mixen, voorbereiden voor een maal instantieren. zoeken of ie er al is shared initialize, bv de selectie weer op all zetten of zoiets window title in menu zetten menu met checkboxen maken]

[to do: opmerking over gebruik change-class niet voor note wel voor window]

Adding behavior to class definitions

Repetitive definitions, constructor of same name and type def change the way the class is defined, by having it not being an instantiation of standard class but of our own meta class

```
(defclass our-class (standard-class)())



(defmethod initialize-instance :after ((object our-class) &rest args

&key name &allow-other-keys)

(make-constructor name name)

(make-type-predicate (intern (string-upcase (format nil "~A?" name))) name))
```

[to do: hoe kemen we hier aan de extra initargs die bv de naam van predicaat en constructor kunnen doorgeven uit de defclass form (defclass note (basic-musical-object)() (:constructor 'note :type-predicate 'note?))]

```
(defun make-constructor (name class)

(when name

(define name

#'(lambda(&rest args)(apply #'make-instance class args)))))



(defun make-type-predicate (name class)

(when name

(define name

#'(lambda(object)

(typep object class)))))
```

```
;(trace define)
```

following warns user redefinition, if confirmed new class with auto type and constructor

```
(defclass note (basic-musical-object)
((pitch :accessor pitch
:initarg :pitch
:initform 60
:documentation "Pitch in MIDI numbers"))
(:documentation "A pitched note")
(:metaclass our-class))
```

should pass initargs name for accessor, visible, but refine left for user (see book MOP)

A defclass form mentioning our meta class instead of defaulting to `standard- class` will result in the creation of a proper type predicate and a constructur function. E.g:

In this way we have effectively extended the CLOS language itself while programming in CLOS. This does not mean that mofdularity and transparencey is violated: our changes to the language cannot affect other parts of the program - only classes defined to a of our metaclass exhibit the new behavior.

We hope that with the above short introduction to the meta object protocol the reader need not be convinced anymore of the power and beauty of CLOS. We will now work on some more mundane issues of object oriented programming, to return later to the this meta-sphere.

Eql specialization, moet voor mouse-click

specialize soms op een bv eql nil afvangen midi note on = 0 modifier keys, click# commandos run :window :object :selected

mop one/few instance classes

use in allocate note, midiport, i/o-format en natuurlijk class cocktail (een tequila sunsire is a tequila sunrise) identiteit

```
(defclass one-instance-only-mixin ()())
```

```
(defvar *one-instance-only-table* (make-hash-table))


(defmethod make-instance :around (class &rest args)

(cond ((null (member 'one-instance-only-mixin

(class-direct-superclasses (find-class class))))

(call-next-method))

((gethash class *one-instance-only-table*))

(t (setf (gethash class *one-instance-only-table*)

(call-next-method)))))


(defclass test1 (one-instance-only-mixin)())

(defclass test2 (one-instance-only-mixin)())


(make-instance 'test1)
```

Standard method combination recap

around before after + diagram + reeds gedaan example beschrijven

build in and/append method combination

Non standard but built in, other ways in which methods are combined e.g. look at same?

```
(defmethod same? ((object1 note)(object2 note))

"Are two notes equal?"

(and (= (pitch object1) (pitch object2))

(= (duration object1) (duration object2))))


(defmethod same? ((object1 pause)(object2 pause))

"Are two rests are equal?"

(= (duration object1) (duration object2)))
```

could not share the fact that for basic musical objects the duratiuons should be the same. Can as around is exercise

But more natural, and combination of results of methods. first undefine. then notify a special method

combination to be used to the generic function

```
(fmakunbound 'same?)

(defgeneric same? (object1 object2)
(:method-combination and))
```

now can share code, and run duration automatically, even for new basic objects to be defined

```
(defmethod same? and ((object1 basic-musical-object)(object2 basic-musical-object))
"Are the two durations are equal?"
(= (duration object1) (duration object2)))

(defmethod same? and ((object1 note)(object2 note))
"Are the two pitches equal?"
(= (pitch object1) (pitch object2)))
```

(same? (note :duration 1 :pitch 50) (note :duration 2 :pitch 50)) -> nil

Other definitions for compounds, only one method for now

```
(defmethod same? and ((object1 sequential)(object2 sequential))
"Are two sequential musical objects equal?"
(same-list (elements object1)(elements object2) :test #'same?))

(defmethod same? and ((object1 parallel)(object2 parallel))
"Are two two parallel musical objects equal?"
(same-set (elements object1)(elements object2) :test #'same?))

(defmethod same? and ((object1 collection)(object2 collection))
"Are two two parallel musical objects equal?"
(and (equal (delta-onsets object1)(delta-onsets object2))
```

```
(same-set (elements object1)(elements object2) :test #'same?)))
```

the default same? method for werkt niet meer need catch type different methods

```
(defmethod same? and ((object1 musical-object)(object2 musical-object))
"Return nil as default case: two musical objects of different type are unequal"
(eql (type-of object1)(type-of object2)))
```

(same? (example)(example)) -> t - (same? (example)(note))

But remember our named mixin,is really wrong upto now

(same? (s “Melody1” (note :duration 1 :pitch “C3”) (note :duration 2 :pitch “D#3”) (note :duration 1 :pitch “C#3”)) (s “Melody2” (note :duration 1 :pitch “C3”) (note :duration 2 :pitch “D#3”) (note :duration 1 :pitch “C#3”))) -> t

[to do: leuk voorbeeld zelfde noten andere naam]

but easy to repair

```
(defmethod same? and ((object1 named-mixin)(object2 named-mixin))
(string-equal (name object1) (name object2)))
```

append build in method combination, na mop class-of

more than and, append, can be use not check attributes but collect descriptions of them e.g. in show. anny later mixins which add attributes to say noets will supply an extra method

```
(fmakunbound 'show)
```

```
(defgeneric show (object)
(:method-combination append)
(:documentation ""))
```

```
(defmethod show append ((object basic-musical-object))
```

```
"Return a list describing the duartion of a musical object"
(list :duration (duration object)))

(defmethod show append ((object note))
(list :pitch (midi-number-to-pitch-name (pitch object))))

(defmethod show append ((object sequential))
"Return a list describing a sequential musical object with its components"
(mapcar #'show (elements object)))

(defmethod show append ((object parallel))
"Return a list describing a parallel musical object with its components"
(mapcar #'show (elements object)))

(defmethod show append ((object collection))
"Return a list describing a parallel musical object with its components"
(loop for delta-onset in (delta-onsets object)
for element in (elements object)
collect delta-onset
collect (show element)))

(defmethod show append ((object musical-object))
(list (class-name (class-of object))))
```

proberen, surprise

```
(example)
->
(((:pitch "C3" :duration 1 note)
(:pitch "D#3" :duration 2 note)
(:pitch "C#3" :duration 1 note) sequential)
(:pitch "G2" :duration 4 note)
parallel)
```

most specific first, but can control, werkt niet

```
(defgeneric show (object)
(:method-combination append :most-specific-first)
(:documentation ""))
(defgeneric show (object)
(:method-combination append :most-specific-last)
(:documentation ""))
```

behavior named just simple method + around voor naam

```
(defmethod show :around ((object named-musical-object-mixin))
"Add the constructor and possibly the name to the showing of the elements"
(if (name object)
(list* (named-constructor object) (name object) (rest (call-next-method)))
(call-next-method)))
```

defined method combination

add-boundaries + margins enlarge

```
(defmethod boundary :around ((object compound-musical-object) onset (window named-
mixin))
"Draw the boundary as side effect of its calculation"
(declare (ignore onset))
(enlarge-rectangle (call-next-method)
(named-margin object window)))
```

```
(defmethod boundary :around ((object compound-musical-object) onset (window nested-
mixin))
"Draw the boundary as side effect of its calculation"
(declare (ignore onset))
(enlarge-rectangle (call-next-method)
(nested-margin object window)))
```

eigenlijk te veel, bedoeld voor margin optellen alle mixin een margin mag me mixen allemaal applicable combination = eigen method

```
(fmakunbound 'boundary)


(defmethod boundary ((object note) onset (window draw-window))

"Return a list of the sides of a graphical representation of a note"

(declare (ignore window))

(list onset ; left

(+ (pitch object) .5) ; top

(offset object onset) ; right

(- (pitch object) .5))) ; bottom


(defmethod boundary ((object musical-object) onset (window draw-window))

"Return nil, boundary of musical object is not known"

(declare (ignore onset))

nil)


(defmethod boundary ((object compound-musical-object) onset (window combine-boundary-
mixin))

"Return a list of the sides of a box enclosing a compound musical object"

(apply #'combine-boundaries

(map-elements-onsets object onset #'boundary window)))


(defmethod boundary :around ((object compound-musical-object) onset (window draw-
window)) ; eigen basic

""

(declare (ignore onset))

(let ((boundary (call-next-method)))

(when boundary

(enlarge-rectangle boundary

(margin object window)))))
```

margin methods, add margin to combine, define

```
(defun add-margin (&rest margins)
(apply #'mapcar #'+ margins))
```

make known

```
(define-method-combination add-margin)
```

define it is the combination way

```
(defgeneric margin (object window)
(:method-combination add-margin)
(:documentation ""))
```

and define methods with that combination

```
(defmethod margin add-margin ((object musical-object) (window nested-mixin))
"The margins of a nested compound boundary box"
(let ((horizontal .15)(vertical .2))
(list (- horizontal) vertical horizontal (- vertical))))

(defmethod margin add-margin ((object compound-musical-object) (window named-mixin))
"Leave space in a boundary box for the name"
(if (name object)
(list 0 .7 0 0)) ; uit window height
'(0 0 0 0))

(draw (example)) =>
```

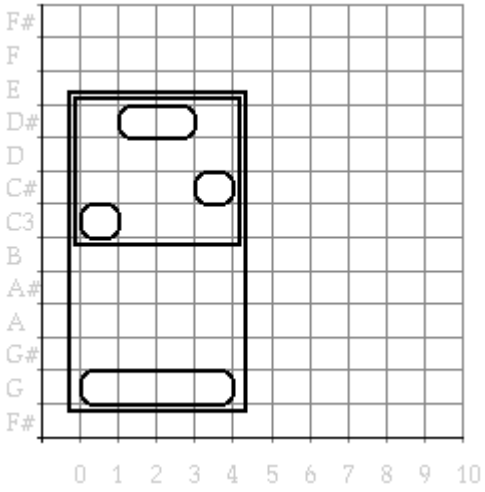

Figure ?. test

[to do: ook voor loudness mixin]

Loudness mixin voor note waar ??
Backpointers

When creating a musical object as part of a compound one, the compound object has access to its parts because these occur in its elements slot. But access the other way: from a musical object to the encompassing compound one is not supported. Such acces is needed to implement, e.g., deletion of a musical object: it has to be removed from the list of elements of its containing compound musical object, if it has one. [10]To program such access, let us first add a slot to a musical object that can contain the reference to a containing compound object.

```
(defclass musical-element ()

((element-of :accessor element-of :initarg element-of :initform nil))

(:documentation "T"))


(defclass musical-object (musical-element)()

(:documentation "The abstract root class of all musical objects"))
```

By supplying an initform value to the slot descriptor, the `element- of` slot will default to `nil` when not specified otherwise. This will be the proper value for a top-most musical structure, which is not an element of anything. But when a compound musical structure is created, incorporating an musical object as its element, this element has to be given a proper backpointer. We will define an after-method for this backpointer initialization, remembering to use a compatible lambda list.

```
(defmethod initialize-instance :after ((object compound-musical-object) &rest
arguments)

"Install backpointers in each component of a compound musical object"

(declare (ignore arguments))

(loop for element in (elements object)

do (setf (element-of element) object)))
```

Now the part-of links between musical objects can be traversed in two ways (see Figure ?).

[to do: backpointers in tree draw program voor het example object, vgl met figure ??]

maintaining back-pointer consistency

never worry, bit expensive later trim. dangling pointers always danger the elements of a compound musical structure change, as we did in the contraction example, the backpointers are in danger to be left dangling ??, i.e. pointing to wrong or deleted musical objects. The backpointer of an object should be realligned to point to the object of which it is an element. Instead of programming this explicitly in each program that modifies the musical structure, it is better to make the primitive modification action, a change of the elements slot, behave such that it will leave a consistent structure. These changes will all be made with the call of a setf on the elements slot of a compound musical structure, and any repair of backpointers can be done in a after method on this setf. In this way all structure modifying transformations will automatically leave consistent backpointers. When storing redundant information is chosen as a method for implementation - as we do for backpointers, but e.g. not for onset times of musical objects - acces function that change the structure are responsible for repairing any inconsistent state that may result. MOET beter in acces

```
(defmethod (setf elements) :before (elements (object compound-musical-object))

(loop for element in (elements object)

do (setf (element-of element) nil)))
```

```
(defmethod (setf elements) :after (elements (object compound-musical-object))

(loop for element in elements

do (setf (element-of element) object)))
```

Destructive operations on the stucture of displayed musical objects

Using this new backpointer facility we can write a method to remove an object: we just remove all the

references to the object in the object's container:

```
(defmethod clear-object ((object musical-object))
(let ((container (element-of object)))
(setf (elements container)
(remove object (elements container)))))
```

We will use this method for `clear- selection` that will suppor clearing selections in draw-windows that have a selection-mixin. It takes care of three situations. The first is no selection at all, then there is nothing to clear. Second, when the whole object is selected (tested with the predicate `all- selected?`) we remove the object from the window and deselect the selection. Finally, the case were a sub-selection is made, we remove only this object using `clear- object` that handles all references correctly.

```
(defmethod all-selected? ((window selection-mixin))
"Return T if selection is the whole musical object, false otherwise"
(eql (selection window) (object window)))

(defmethod no-selection? ((window selection-mixin))
"Return T if there is no selection, false otherwise"
(eql (selection window) (object window)))

(defmethod clear-selection ((window selection-mixin))
"Clear selection from window"
(cond ((no-selection? window) nil)
((all-selected? window)
(setf (object window) nil)
(deselect window))
(t (clear-object (selection window)))))

(draw (example)) =>
```

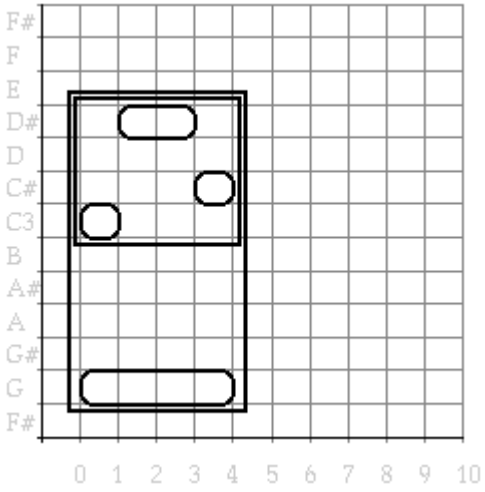

Figure ?. A. Clearing a selection.

```
(front-mouse-click 1 58 nil 1) =>
```

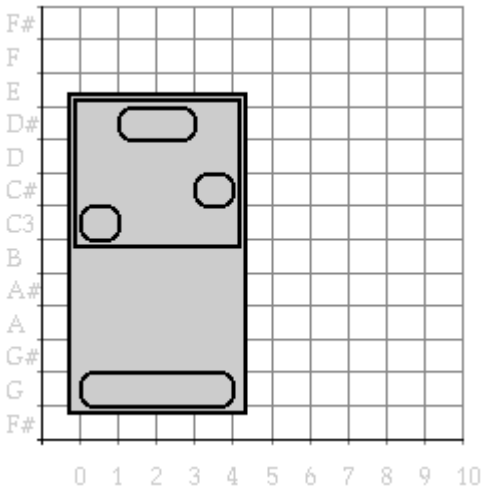

Figure ?. B. Clearing a selection.

```
(do-front-window #'clear-selection)

(front-draw-window) =>
```

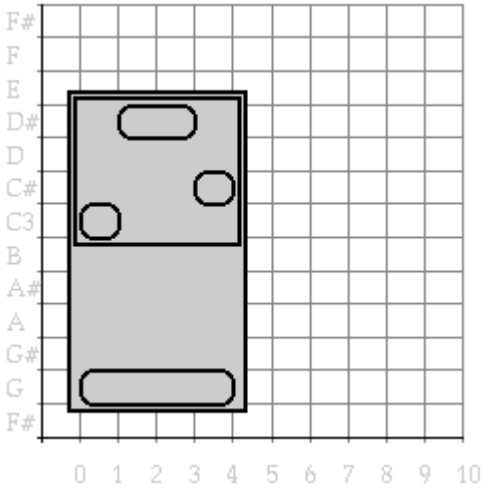

Figure ?. C. Clearing a selection.

The other situation in which we have to keep the backpointers consistent is when we want to replace a selection with another music-object. For this we write a `replace- object` function that replaces an musical-object (the first argument) with another (the second argument). The method replace-selection uses it.

```
(defmethod replace-object ((orginal musical-object)(replacement musical-object))

"Return object with replacement, adjusting backpointer accordingly"

(loop for element in (elements (element-of orginal))

when (eql element orginal)

collect replacement

else collect element))


(defmethod replace-selection ((window selection-mixin)(object musical-object))

"Replace selection in window with musical-object"

(cond ((no-selection? window) nil)

((all-selected? window)

(setf (object window) object)

(deselect window))

(t (replace-object (selection window) object))))
```

Furthermore, we can write transformations that work appropiately on selections.

```
(defmethod transform-selection ((window selection-mixin) transform &rest args)
(when (selection window)
(replace-selection window (apply transform (selection window) args))))

(defmethod repeat-selected ((window selection-mixin) &optional (n 2))
(transform-selection window #'repeat))

(defmethod canon-selected ((window selection-mixin))
(transform-selection window #'canon))



(add-music-command "Canon selected" #'canon-selected #\C)
(add-music-command "Repeat selected" #'repeat-selected #\R)

(draw (example)) =>
```

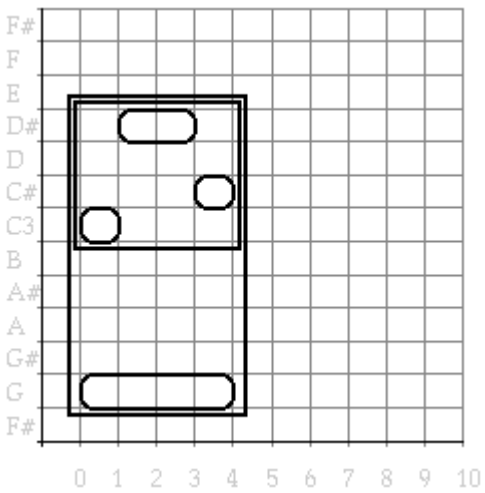

Figure ?. A. Transforming a selection.

```
(front-mouse-click 1 58 nil 1) =>
```

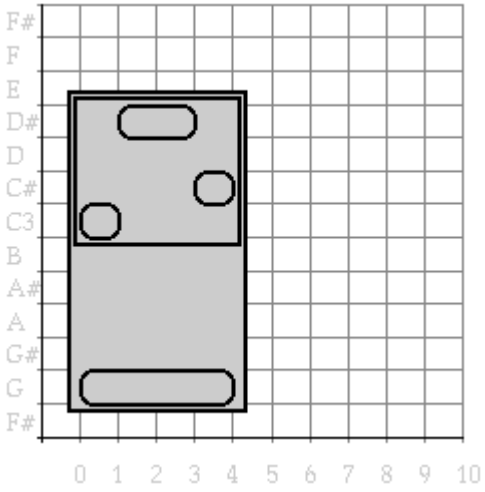

Figure ?. B. Transforming a selection.

```
(do-front-window #'repeat-selected)

(front-draw-window) =>
```

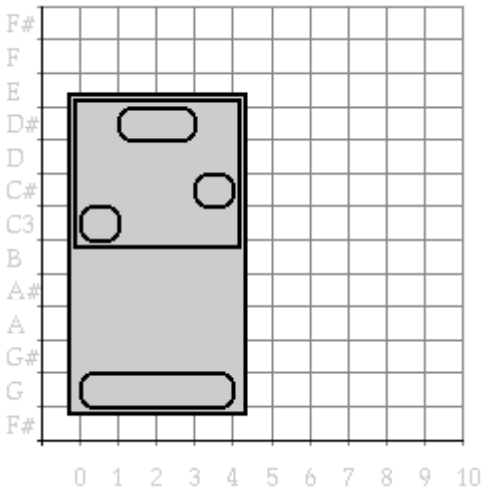

Figure ?. C. Transforming a selection.

[to do: menu extra keywords? enablep op window, op selectie, op non-selecte, ...]

A clipboard for musical objects

We now can make different kind of selections (`clear- selection`, `replace- selection`, etc), in a programmed fashion (using,e.g., :lisp do-front-window

) and directly on the window, with mouse clicks and command-keys. Now assume we have two of these windows. Than we need a way to transfer this information to another window. When programming, we can bind the objects (e.g., in a `let`), when using control keys we need another way of (temporary) storing information. The standard Macintosh way to do this is via the clipboard, a data structure containing information that was cut or copied (using the standard Cut and Copy edit commands) and to make it available for pasting somewhere else (You probably use it all the time for editing text when writing code in your editor). The functions `put- scrap` and `get- scrap` are available for this. [11] communicate information to the clipboard. They need the type of the information and, in the case of `putscap`, the data to put on the clipboard.

```
(put-scrap :text "A string of text.") ->

"A string of text."
```

```
(get-scrap :text) ->

"A string of text."
```

We now want to make copying, pasting and cuting of musical-objects from and to windows as easy as text copying is. For this to work, we have to tell the system about musical-objects as a new type that can be handled by the clipboard and define our own scrap-handler (the program that deals with clipboard information). We define first are own class `clipboard`, inheriting from (MCL's) `scrap- handler`. Then we define our own specialized :lisp musical-object-clipboard

that can deal with musical-objects. An initialize-instance after-method tells the system about this new scrap-handler, using the default initialization argument. **[to do: import`ccl::add- scrap- handler` ?, uitleg after default-initargs eerder al gedaan hebben]**

```
(defclass clipboard (scrap-handler) ()

(:documentation "scrap-handler of for musical-objects"))
```

```
(defclass musical-object-clipboard (clipboard) ()

(:default-initargs :scrap-type :musical-object)

(:documentation "scrap-handler of type :musical-object"))
```

```
(defmethod initialize-instance :after ((object clipboard) &rest args &key scrap-type)

(ccl::add-scrap-handler scrap-type object))
```

```
(make-instance 'musical-object-clipboard)
```

We can now get and put our musical-objects from and to the clipboard. Note that true musical-objects are

put on the clipboard, not just their printed represention.

```
(put-scrap :musical-object (note)) ->

(note :duration 1 :pitch "C3")


(get-scrap :musical-object) ->

(note :duration 1 :pitch "C3")
```

Standard edit operations on window with musical objects

Of course, we want use the musical-object clipboard type with the standard Macintosh edit facilities. For example, copy an object from a window and paste it in another. First, we will write a set of edit functions, using the previously defined selection functions and our new clipboard facility for musical-objects.[12]

```
(defmethod copy ((window selection-mixin))

"Set scrap to current selection in window"

(put-scrap :musical-object (selection window)))


(defmethod clear ((window selection-mixin))

"Remove current selection from window"

(clear-selection window))


(defmethod cut ((window selection-mixin))

"Set scrap to current selection and remove selection from window"

(put-scrap :musical-object (selection window))

(clear-selection window))


(defmethod paste ((window selection-mixin))

"Paste musical-object from scrap in window"

(replace-selection window (get-scrap :musical-object)))
```

Second, we have to tell our draw-window if, and in what situation these operations are allowed. The Lisp system checks whether the common macintosh edit actions (like `select- all`) are supported for a specific window that happens to be the front window, and in that case the corresponding menu item, and the corresponding command key are enabled. By writing a `window- can- do- operation` method specifically for the `selection- mixin`, we can use,for example, the command-C keystroke and the “Copy” menu item under the edit menu on any window whose class is based on the `selection- mixin`.

This is a way in which system code was prepared to be extended: it uses a method which is defined for the genaral default case (`window- can- do- operation` returns :lisp nil

for arbitrary windows), and any user-code can define specialized methods for it. Below we define the method for the four stand edit operations. Check the situations in which they are enabled:

```
(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'copy))

&optional item)

"Can copy when window has a musical-object selected"

(selection window))


(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'clear))

&optional item)

"Can clear when window has a musical-object selected"

(selection window))


(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'cut))

&optional item)

"Can cut when window has a musical-object selected"

(selection window))


(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'paste))

&optional item)

"Can paste when a musical-object is on the clipboard"

(get-scrap :musical-object))
```

Before we will try our newly defined edit operations, we will add another function to our music -menu: a function to make a new and empty piano-roll window: **[to do: add-music-command needs extra arg: menu-item should always be enabled]**

```
(defmethod new-draw-window ((window selection-mixin) &rest args)
```

```
(apply #'draw nil args))
```

```
(add-music-command "New Draw Window" #'new-draw-window #\N)
```

Now we will make two windows, one with our familiar example, the other empty.

```
(draw (example) :window-title "draw window A") =>
```

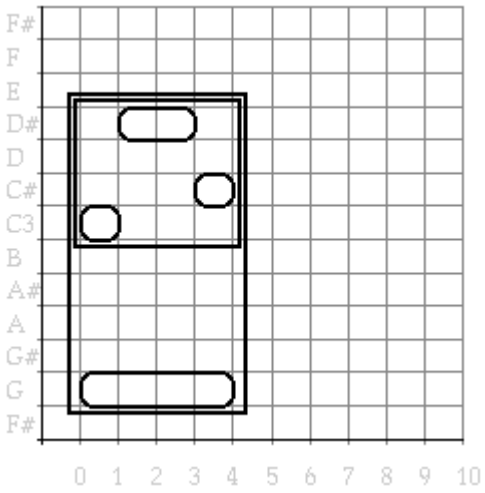

Figure ?. A. Make two windows

```
(draw nil :window-title "draw window B" :view-position (make-point 250 40)) =>
```

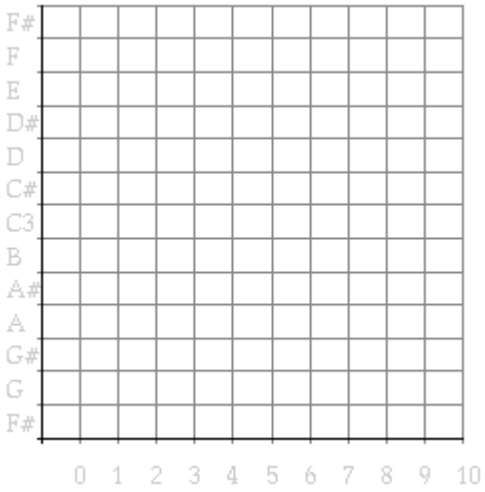

Figure ?. B. Make two windows

In order to switch, in a programmed way (running the code you can of course select one or the other window by clicking on it), between the one and the other draw-window we define the following function:

```
(defun second-draw-window ()
"Return the second frontmost drawing window, bringing it to the front as well"
(let ((window (second (windows :class 'draw-window))))
(when window
(window-select window)
window)))
```

In the next figures we can see how we can use our newly defined edit operations.

```
(second-draw-window)
(do-front-window #'select-all)
(front-draw-window) =>
```

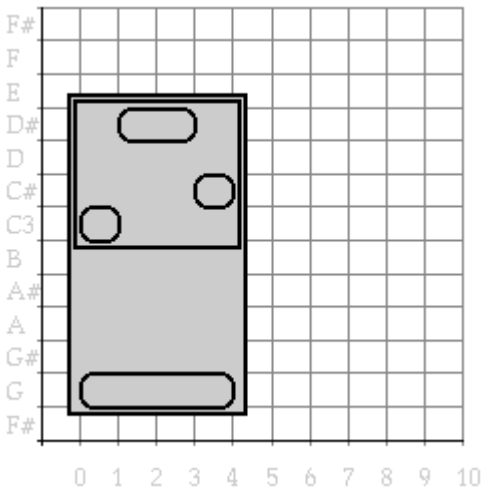

Figure ?. A. Edit opereration on windows

```
(do-front-window #'copy)
```

```
(front-draw-window) =>
```

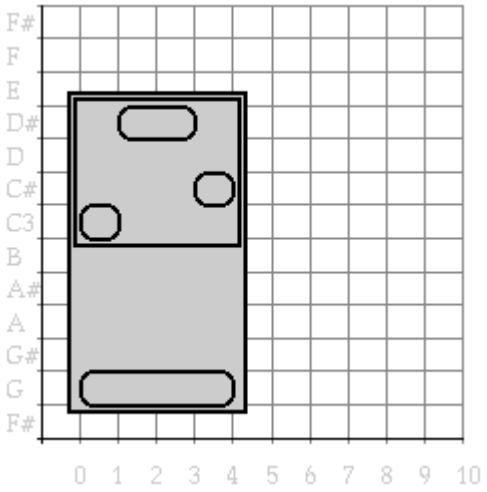

Figure ?. B. Edit opereration on windows

```
(second-draw-window)

(do-front-window #'paste)

(front-draw-window) =>
```

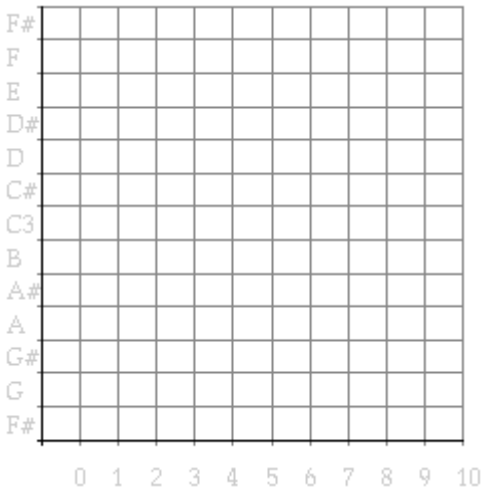

Figure ?. C. Edit opereration on windows

If you start combining these methods with transformation you will soon find out that our idea of copying

did not really work. We copied a reference co musical-object to the scrap and, consequently in to the other window. When (destructively) changing the musical object in window A, it will also change in window B.

```
(second-draw-window)

(front-window-operation #'transpose 3)

(front-draw-window) =>
```

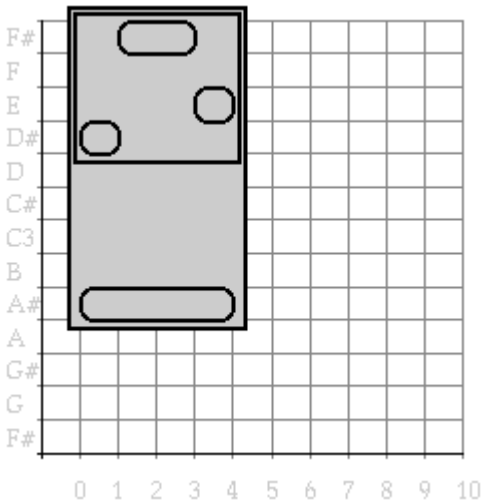

Figure ?. A. Unforeseen side-effects

```
(second-draw-window)

(do-front-window #'select-all)

(front-draw-window) =>
```

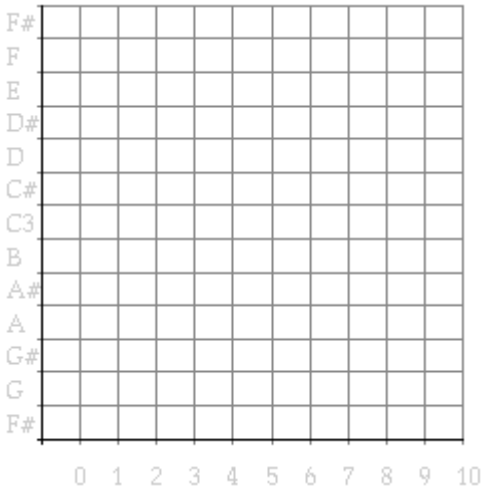

Figure ?. B. Unforeseen side-effects

To prevent this from happening we will copy the object before putting it on the clipboard.

```
(defmethod copy ((window selection-mixin))
"Set scrap to current selection in window"
(put-scrap :musical-object (copy-object (selection window))))

(defmethod cut ((window selection-mixin))
"Set scrap to current selection and remove selection from window"
(put-scrap :musical-object (copy-object (selection window)))
(clear-selection window))
```

But, also when pasting in different windows, each possibly transforming this object again, we need to make a copy from the object when obtainign it from the scrap (which actually is a bug in the clipboard system: it is not an object that is copes, but just refered to.copy but refering that is done).[13]

```
(defmethod paste ((window selection-mixin))
"Paste musical-object from scrap in window"
(replace-selection window (copy-object (get-scrap :musical-object))))
```

Open musical objects

We will integrate the edit operation on musical object a little bit further in the Macintosh environment by providing a open-selection musical-object. We add it as “Open Selection” menut item under our Music menu, and make it available as the action to be done when doublke-clicking on a msucial-object: it will open a new window (of the same class and related title) containing only the selection. Note, that for now, `open- object` copies musical objects, as such updates on the master window will not affect the contents of the newly opened window (for the same reason described for clipboards). [14]

```
(defmethod open-selection ((window selection-mixin))
(open-object (selection window)
:window-class (class-of window)
:window-title (format nil "Part of ~A" (window-title window))
:view-position (add-points (make-point 20 20) (view-position window))))
```

```
(defmethod open-object (object &rest args)

(when object (apply #'draw (copy-object object) args)))



(defmethod mouse-click ((window selection-mixin) time pitch (modifiers (eql nil))
(count (eql 2)))

(select-cover window time pitch)

(open-selection window))



(add-music-command "Open Selection" #'open-selection #\O)
```

[to do: add #'selection als enable-p function: adapt `install- file- type- sub- menu`]

[to do: als de auto domain schaling gemaakt is moet openen van een deel zinnig zijn: uitgezoomed plaatje, nu foot note: how would you design this?]

draw rests

rusten verbieden in P en collectie (around setf en after init) en s mag niet alleen uit een rust bestaan dan makkelijk te tekenen

After creating a compound-musical-object all its elements will have a proper backpointer to it. As an example of the use of this construct consider the drawing of pauses. Uptill now these were simply ignored in drawing. But when a pause is part of a sequential structure, a white box with the height of the enclosing box would make a good graphical representation, and it would make the draw program more complete. footnote A rest in a parallel structure, or on the top level does not make much sense anyhow and could be ignored.

We only have to define a boundary method for pauses to achieve that. And because the calculation of the pitch range of the enclosing box for a rest needs accces to the enclosing object, the back-pointers come in really handy now. Remember that we can now at once define a draw-object method for any musical object, because boundary works for each of them.

```
(defmethod boundary ((object pause) onset window)

(when (element-of object)

(let ((boundary (boundary (element-of object))))

(when boundary

(destructuring-bind (left top right bottom) boundary

(list onset to (offset object onset) bottom))))))
```

```
(defmethod draw-boundary ((object pause) boundary window)

(draw-rectangle boundar window 1))
```

hoe oinderscheid maken met noten ??

enlarge boundaries eruit halen

Score system
Sound

steeds aan de lezer vragen dan aan eind helemaal bouwen defclass sound duration file pitc sound? same? loudness name after int probe-file transpose/mirror stretch play csound cut/copy/paste

Mop protocol

abstract class, at make-instance time, no need for mop

```
(defclass abstract-class (standard-class)())


(defmethod make-instance :around ((object abstract-class) &rest args)

(error "Cannot instantiate abstract class ~A" (class-name object)))


(defclass draw-window-mixin ()()

(:metaclass abstract-class))


(make-instance 'draw-window-mixin)

=>

> Error: Cannot instantiate abstract class draw-window-mixin

> While executing: #<standard-method make-instance :around (abstract-class)>
```

mix checking, at defclass time

```
(defclass inheritance-checker (meta-class)

(:before-when-present etc slots to be filled from initargs ??))
```

```
(defmethod check ((object meta-class) tree) t)

(defmethod check ((object inheritance-checker) tree)
(get-init-args object
:before-when-present
:should-be-before
:after-when-present
:should-be-after
:should-be-with
:may-not-be-with))

; reasons meegeven

(defmethod check-inheritance ((object inheritance-checker) list tree)
(loop for class in list
unless (check class tree)
do (error "Inheritance violation in class ~A, detected by ~A"
(name-of object) (name-of class))))

(defmethod initialize-instance :around ((object inheritance-checker) &rest args)
(let ((class (call-next-method)))
(check-inheritance class (inheritance-list class) (inheritance-tree class))
class))
```

should-have methods instatiation time but needs class, object may not be made

```
(defclass method-checker (meta-class)
(needed-methods))

(defmethod make-instance :before ((object method-checker) &rest args)
(unless (check-methods object)
(error "Cannot instantiate unfinished class ~A" (name-of object))))
```

```
(defmethod check ((object inheritance-checker) tree)

(check all needed methods present?))
```

vb, visible-structure needs calc-boundaries

auto mixin

```
(defclass auto-mixin (meta-class)

(conditions))
```

```
(defclass allow-auto-mixin (meta-class)

)
```

dan voor aanmaken allow-auto-mixins even alle auto-mixins proberen, als conditie dan toevoegen aan direct-superclasses, altijd vooraan

mop cocktail methods

als specializer and or not dan in compute applicable methods die ook terug geven of, als truc anonieme classen aanmaken die substitueren in method die classen automaties in mixen, in die specifieke gevallen around en zelf invocatie doen compute applicable methods doen

init keyword class adder

meta class instance init-mixer before make-instance van het object zelf, classes bijmixen (als uitbreiding van boven?)

one instance class

voor formats bv MTX en CSOUND gebruik in ??, eerst via table/resource noot allocatie, only 16 midi poort

als mop gebruik abstracte classes

Full code listing

problems in presentation order classes/objects load restrictions too much freedom as ADT class + methods, but not multi multimethods problem no real solution but browsers, progr environment conceptual order, logical order of domain good exercise present final version plus comment style/headers etc so much redefinitions, not much code

It is difficult to find a good order to present object oriented code, and it is often argued that this is no longer needed, given the existence of code browsers and other tools in the programming environment, and although these tools are indispensable, it is worth while to find a nice order, and add some comments at the level of groups of related programs etc. Other programmers that have to read your code, and even yourself (some time after you wrote it), will be thankful for a good listing. That is why we present in the appendix all code developed in this chapter. It is re-ordered and only the last versions of functions are kept. The listing contains some details that were omitted in the text (on handling pitch names and low-level byte manipulations for writing MIDI files). In the real world the code would have been split in a couple of files - but we present them here as one. Separation lines and comments at different levels indicate the structural units within the program.

Conclusion

[to do: beter uitwerken recap en gebruik vertellen]What have we got after all this writing and rewriting of code? We have designed a data representation for musical objects that proved quite flexible. We wrote tools for printing, drawing, playing and outputting them in different formats. We have defined a couple of transformations on them. The modularity is of a very fine grain - most programs are a couple of lines long, all take less then 10 lines of code. Though the code achieves quite some complex behavior, e.g. the nested drawing of boxes around compound musical objects or the writing of standard MIDI files, it still is easy to oversee and maintain because of that modularity. However, it has to be stressed that most programs are still bare skeletons of what a complete computer music system needs. But they are very good skeletons, and easy to fill-in. The constructions used and explained are the basic, object oriented constructs of CLOS. However, even these constructs (take e.g. the method combination for around methods) surpass with ease the power found in other object oriented languages. The design of programming language environments for the time based arts is so complicated in itself that programming language constructs that help in managing the complexity become a crucial issue NIL

. We hope to have shown how the constructs of CLOS can help achieving this.

Definitions made

`collection (compound-musical-object) class`

A group of musical objects, starting at specified times

`nested-mixin (draw-window-mixin) class`

A mixin to draw the structure in a nested fashion

`named-mixin (draw-window-mixin) class`

A mixin to print name tags of objects

`midi-awareness-mixin (draw-window-mixin) class`

A mixin to show distinction between MIDI and free notes

`compound-musical-object (musical-object named-musical-object-mixin) class`

A named group of musical objects ordered in time

`named-musical-object-mixin nil class`

A named object

`midi-note-mixin nil` *class*

A note mixin which will guarantee an integer pitch number

`midi-note (midi-note-mixin note)` *class*

A note which will guarantee an integer pitch number

`map-musical-object-onsets (object onset combination operation &rest args)` *method*

Apply the operation tho the object and its onset

`mapc-musical-object (object operation &rest args)` *method*

Apply the operation to this object

`mapc-musical-object-onset (object onset operation &rest args)` *method*

Apply the operation to this object, and to each sub-..-sub-object and their onsets

`get-factor nil` *function*

Ask user to type a factor and return it as positive number

`get-interval nil` *function*

Ask user to type an interval in semitones and return it as number

`get-pitch nil` *function*

Ask user to type an pitch name

`(setf pitch) (value object)` *method*

Call the primary method only with a pitch that is kept within [0,127]

`(setf pitch) (value object)` *method*

Call the primary method with a pitch number

`(setf pitch) (value object)` *method*

Call the primary method with a pitch that is kept within [0,127]

`window-can-do-operation (window operation &optional item)` *method*

Can paste when a musical-object is on the clipboard

`clear-selection (window)` *method*

Clear selection from window

`transpose (object interval)` *method*

Don’t transpose when this would move the pitch outside the MIDI range

`draw-musical-object (object onset window)` *method*

Draw a graphical piano-roll representation of a musical object

`draw-musical-part (object onset window)` *method*

Draw a graphical representation of the musical object itself

`margin (object window)` *method*

Leave space in a boundary box for the name

`named-margin (object window)` *method*

Leave space in a boundary box for the name

`combine-event-lists (object onset self parts)` *method*

Merge a list of sorted agenda's into one sorted agenda

`merge-events (&rest events)` *function*

Merge a list of sorted agenda's into one sorted agenda

`multf (args)` *macro*

Multiply a generalized variable

`stretch (object factor)` *method*

Multiply the duration of a musical object by a factor

`paste (window)` *method*

Paste musical-object from scrap in window

`mirror-selection (window)` *method*

Reflect the musical object selected in the window around a center

`clear (window)` *method*

Remove current selection from window

`replace-selection (window object)` *method*

Replace selection in window with musical-object

`all-selected? (window)` *method*

Return T if selection is the whole musical object, false otherwise

`no-selection? (window)` *method*

Return T if there is no selection, false otherwise

`object-note-list (object onset)` *method*

Return a basis event list consisting of one onet-part pair

`collection (&rest agenda)` *function*

Return a collection of musical objects starting at specified onsets

`show (object)` *method*

Return a list describing a compound musical object with its components

`note-list (object onset) method`

Return a list of onset-event pairs of all parts of a musical object

`event-list (object onset object-event-list-method) method`

Return a list of onset-event pairs of all parts of a musical object

`c (name &rest agenda) function`

Return a named collection

`p (name &rest elements) function`

Return a named parallel musical object

`s (name &rest elements) function`

Return a named sequential musical object

`sorted-note-list (object onset) method`

Return a note list: (onset note) pairs, with onsets in ascending order

`note (&rest attributes) function`

Return a note object with specified attributes

`example-collection nil function`

Return a simple collection of 5 notes

`event-merge (list1 list2) function`

Return a sorted event list of all events of both arguments

`replace-object (orginal replacement) method`

Return object with replacement, adjusting backpointer accordingly

`collection? (object) method`

Return t if the object is an instantiation of class collection, nil otherwise

`duration (object) method`

Return the duration of a collection

`second-draw-window nil function`

Return the second frontmost drawing window, bringing it to the front as well

`all-class-subclasses (class) function`

Return the tree of classes inheriting from this one

`class-subclasses (class) function`

Return the tree of classes inheriting from this one

`clip (min value max) function`

Return the value, clipped within [min,max]

`named-example nil` *function*

Returns a simple musical object with named structure

`retrograde (object)` *method*

Reverse all components of the object in time

`cut (window)` *method*

Set scrap to current selection and remove selection from window

`copy (window)` *method*

Set scrap to current selection in window

`(setf duration) (value object)` *method*

Set the duration of a compound musical object by stretching its components

`(setf duration) (value object)` *method*

Set the duration of a compound musical object by stretching its components

`(setf onsets) (onsets object &optional onset)` *method*

Set the onset times of a collection, relative to the onset of the object

`sort-event-list (event-list)` *function*

Sort a list of onset-object pairs in order of increasing onsets

`stretch-selection (window &optional factor)` *method*

Stretch the duration of the musical object selected in the window by a factor

`musical-element nil` *class*

T

`musical-object (musical-element)` *class*

The abstract root class of all musical objects

`nested-margin (object window)` *method*

The margins of a nested compound boundary box

`draw-window (midi-awareness-mixin named-mixin mouse-select-mixin nested-mixin structured-mixin visible-selection-mixin piano-roll-contents-draw-window)` *class*

The window class named pianoroll

`transpose-selection (window &optional interval)` *method*

Transpose the musical object selected in the window by an interval

`make-instance (object &rest args)` *method*

`abstract-class (standard-class)` *class*

draw-boundary (object boundary window) *method*

boundary (object onset window) *method*

mouse-click (window time pitch modifiers count) *method*

open-object (object &rest args) *method*

open-selection (window) *method*

new-draw-window (window &rest args) *method*

initialize-instance (object &rest args &key scrap-type) *method*

canon-selected (window) *method*

repeat-selected (window &optional n) *method*

transform-selection (window transform &rest args) *method*

clear-object (object) *method*

(setf elements) (elements object) *method*

(setf elements) (elements object) *method*

add-margin (&rest margins) *function*

same? (object1 object2) *method*

make-type-predicate (name class) *function*

make-constructor (name class) *function*

our-class (standard-class) *class*

window-change-class (object &optional class) *method*

get-window-cocktail nil *function*

select-direct-subclasses (class) *function*

make-cocktail-instance (ingredients &rest args) *function*

define-cocktail-class (ingredients) *function*

unmidi-selection (window) *method*

midi-selection (window) *method*

change-note-class (object class) *method*

intonation-example nil *function*

update-instance-for-different-class (old new &rest initargs) *method*

(setf pitch) (value object) *method*

midi-note? (object) *function*

midi-note (&rest args) *function*

named (name) *function*

named-constructor (object) *method*

enlarge-rectangle (box margins) *function*

`onsets (object &optional onset)` *method*

`clipboard (scrap-handler)` *class*

scrap-handler of for musical-objects

`musical-object-clipboard (clipboard)` *class*

scrap-handler of type :musical-object

Literature references made

(Pope, 1992)

Balaban, M. 1992 Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music. in Balaban, M., K. Ebcioglu & O. Laske (Eds.) Cambridge, Mass.: MIT Press. 111-138

Chowning, 1973 Journal of the Audio Engineering Society, 21(7):526-534. Reprinted inC. Roads and J. Strawn, (Eds.) 1985 Foundations of Computer Music. Cambride, MA: MIT Press. pp. 6-29.

Dannenberg, R.B. 1991 Real-time Scheduling and Computer Accompaniment. in Mathews, M.V and J.R. Pierce (Eds.) . Cambridge, MA: MIT Press.

Dannenberg, R.B. 1993 The Implementation of Nyquist, a Sound Synthesis Language. in San Francisco: ICMA.

Dannenberg, R.B., P. Desain and H. Honing (in Press) Programming Language Design for Music. in G. De Poli, A. Picialli, S.T. Pope & C. Roads (Eds.) Lisse: Swets & Zeitlinger.

Desain, P. and H. Honing (In Preparation) CLOSe to the edge, Advanced Object-Oriented Techniques in the Representation of Musical Knowledge. Jounral of New Music Research.

Desain, P. 1990 Lisp as a second language, functional aspects. (28)1 192-222

International MIDI Association 1983 North Holywood: IMA.

Moore, F.R. 1990 Englewood Cliffs, New Jersey: Prentice Hall.

“Dannenberg, Desain and Honing (in Press)”

Pope, S.T. 1992 The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers.16(3) 73-91

Pope, S.T. 1993 Machine Tongues XV: Three Packages for Software Sound Synthesis. 17(2) 23-54

Steele, G.R. 1990 Bedford MA: Digital Press.

Tatar,D.G. 1987 Bedford, MA: Digital Press.

Glossary references made

abstract class

abstract class

access function

all functions part of a data abstraction layer (selector and constructor functions). [loaded from Glossary Functional]

accessor function

after method

anonymous function

A function whose ‘pure’ definition is given, not assigning it a name at the same time. [loaded from Glossary Functional]

applicable method

applicable methods

application

obtaining a result by supplying a function with suitable arguments. [loaded from Glossary Functional]

assignment

atom

in Lisp: any symbol, number or other non-list. [loaded from Glossary Functional]

before method

class

class hierarchy

class options

combinator

A function that has only functions as arguments and returns a function as result. [loaded from Glossary Functional]

compile time

congruent lambda lists

cons

A Lisp primitive that builds lists. Sometimes used as verb: to add an element to a list. [loaded from Glossary Functional]

constant function

A function that always returns the same value [loaded from Glossary Functional]

constructor function

A function that as part of the data abstraction layer provides a way of building a data structure from its components. [loaded from Glossary Functional]

constructor functions

continuations

A way of specifying what a function should do with its arguments. [loaded from Glossary Functional]

coroutines

parts of the program that run in alternation, but remember their own state of computation in between switches. [loaded from Glossary Functional]

data abstraction

A way of restricting access and hiding detail of data structures [loaded from Glossary Functional]

data abstraction

data type

A class of similar data objects, together with their access functions. [loaded from Glossary Functional]

declaration

declarative

declarative method combination

dialect

A programming language can be split up into dialects that only differ (one hopes) in minor details. Lisp dialects are abundant and may differ a lot from each other even in essential constructs. [loaded from Glossary Functional]

direct superclasses

dynamically typed language

effective method

first class citizens

rule by which any type of object is allowed in any type of programming construct. [loaded from Glossary Functional]

free variables

function

A program or procedure that has no side effects [loaded from Glossary Functional]

function composition

the process of applying one function after another. [loaded from Glossary Functional]

function quote

A construct to capture the correct intended meaning (with respect tothe current lexical environment) of a anonymous function so it can beapplied later in another environment. It is considered good programming style to use function quotes as well when quoting the name of a function. [loaded from Glossary Functional]

functional abstraction (or procedural abstraction)

A way of making a piece of code more general by turning part of it into a parameter, creating a function that can be called with a variety of values for this parameter. [loaded from Glossary Functional]

functional argument (funarg)

A function that is passed as argument to another one (downward funarg) or returned as result from other one (upward funarg) [loaded from Glossary Functional]

generalizedvariable

generic

global variables

an object that can be referred to (inspected, changed) from any part of the program. [loaded from Glossary

Functional]

higher order function

A function that has functions as arguments. [loaded from Glossary Functional]

imperative style

inheritance

initializationkeyword

initializationprotocol

initialization argument list

initialization keyword

instance

instantiation

iteration

repeating a certain segment of the program. [loaded from Glossary Functional]

lambda list keyword

lambda-list keyword

A keyword that may occur in the list of parameter names in a function definition. It signals how this function expects its parameters to be passed, if they may be ommited in the call etc. [loaded from Glossary Functional]

lexical scoping

A rule that limits the 'visibility' of a variable to a textual chunk of the program. Much confusion can result from the older- so called dynamic scoping - rules. [loaded from Glossary Functional]

list

message passing

message passing style

meta object protocol

method

method combination

The declaritive way in which CLOS allows more methods to be bundled and run, in situations where more are applicable [loaded from Object Oriented I]

method combination

method qualifier

methods

mixin class

modification macro's

most specific method

multi-methods

multi-methods

multiple inheritance

multiple inheritance

name clash

name clashes

object oriented programming

A style of programming whereby each data type is grouped with its own access function definitions, possibly inheriting them from other types. [loaded from Glossary Functional]

object-oriented style

parameter-free programming

A style of programming whereby only combinators are used to build complex functions from simple ones. [loaded from Glossary Functional]

part-of hierarchy

polymorphic

polymorphism

prefix notation

A way of notating function application by prefixing the arguments with the function. [loaded from Glossary Functional]

pretty printing

primary method

primary methods

procedural

quote

A construct to prevent the Lisp interpreter from evaluating an expression. [loaded from Glossary Functional]

read-eval-print

record

recursion

A method by which a function is allowed to use its own definition. [loaded from Glossary Functional]

run time

selector function

A function that as part of the data abstraction layer provides access to a data structure by returning part of it. [loaded from Glossary Functional]

selector functions

setf method

setf methods

shadow

side effect

Any actions of a program that may change the environment and so change the behavior of other programs. [loaded from Glossary Functional]

side effect

slot descriptors

slots

stack

A list of function calls that are initiated but have not yet returned a value. [loaded from Glossary Functional]

standard method combination

statically typed

structure preserving

A way of modifying data that keeps the internal construction intact but may change attributes attached to the structure. [loaded from Glossary Functional]

tail recursion

A way of recursion in which the recursive call is the ‘last’ thing the program does. [loaded from Glossary Functional]

the most specific method

untyped language

[an error occurred while processing this directive]